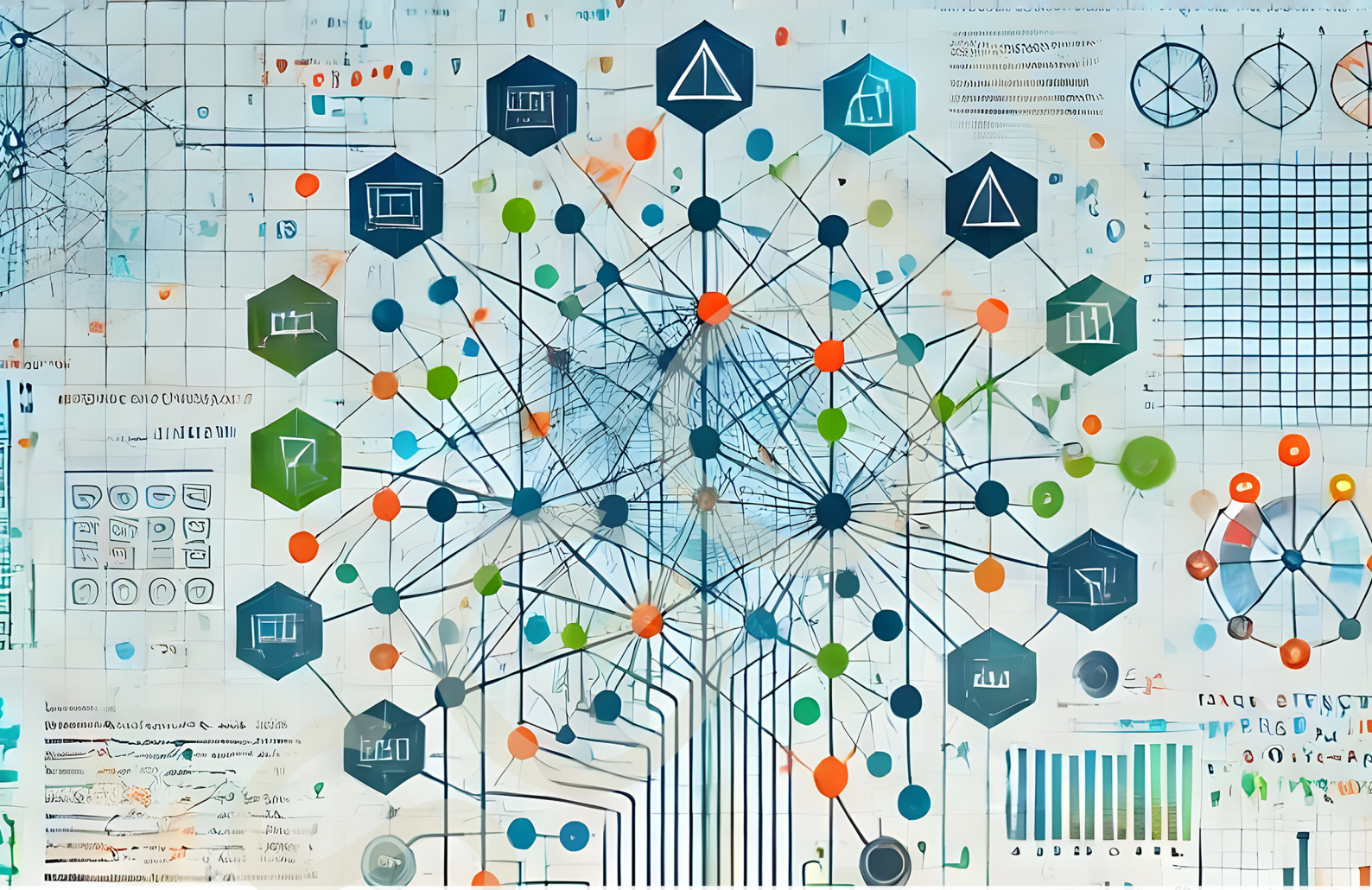
Data Structures

COURSE CODE: B21CA04DC

Bachelor of Computer Applications

Discipline Core Course



SELF LEARNING MATERIAL

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Data Structures

Course Code: B21CA04DC

Semester - II

Discipline Core Course
Undergraduate Programme
Bachelor of Computer Applications
Self Learning Material

(With Model Question Paper Sets)

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

Data Structures
Course Code: B21CA04DC
Semester - II
Discipline Core Course
BCA

SREENARAYANAGURU
OPEN UNIVERSITY

Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.
www.sgou.ac.in

ISBN 978-81-978764-7-9
9 788197 876479

DOCUMENTATION

Production

September 2024

YouTube

MESSAGE FROM VICE CHANCELLOR

Dear

With immense joy and excitement, I extend my heartfelt greetings to all of you and warmly welcome you to Sreenarayanaguru Open University.

Established in September 2020 as a state-driven initiative, Sreenarayanaguru Open University is dedicated to advancing higher education through open and distance learning. Our vision is guided by the principle of "access and quality define equity," laying the foundation for a celebration of excellence in education. I am delighted to share that we are steadfast in our commitment to uphold the highest standards and refrain from compromising on the quality of education we offer. The university draws its inspiration from the legacy of Sreenarayana Guru, a revered figure in the Indian renaissance movement. His name serves as a constant reminder for us to prioritize quality in all our academic endeavors.

Sreenarayanaguru Open University operates within the practical framework of the widely recognized "blended format." Acknowledging the constraints faced by distance learners in accessing traditional classroom settings, we have curated a pedagogical approach centered on three main components: Self Learning Material, Classroom Counselling, and Virtual Modes. This comprehensive blend is poised to deliver dynamic learning and teaching experiences, maximizing engagement and effectiveness. Our unwavering commitment to quality ensures excellence across all aspects of our educational initiatives.

The university aims to offer you an engaging and stimulating educational environment that fosters active learning. The SLM is designed to offer a comprehensive and cohesive learning experience, fostering a deep interest in the study of technological advancements in IT. Careful consideration has been given to ensure a logical progression of topics, facilitating a clear understanding of the discipline's evolution. The curriculum is thoughtfully crafted to provide ample opportunities for students to navigate through the current trends in information technology. Furthermore, this course is designed to provide essential insights into computer hardware, software classification, and foundational HTML concepts crucial for web development.

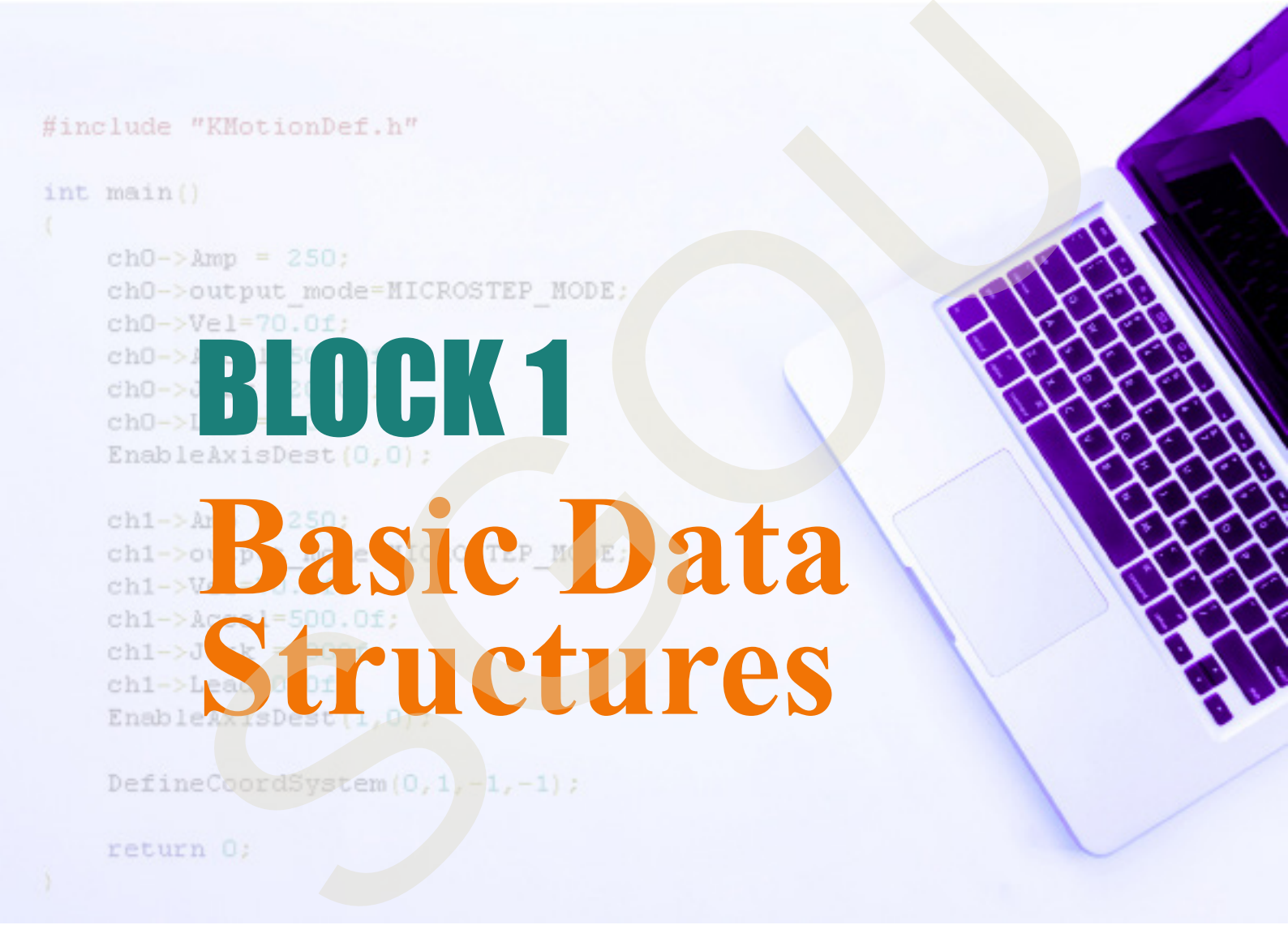
We assure you that the university student support services will closely stay with you for the redressal of your grievances during your studentship. Feel free to write to us about anything that seems relevant regarding the academic programme.

Wish you the best.

Regards,
Dr. Jagathy Raj V. P.

24-04-2024

Contents



BLOCK 1

Basic Data Structures

UNIT 1

Linear and Nonlinear Structures

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- familiarise the concept of Data and Information
- identify the need for various data types and data structures
- explain the classification of data structures.
- make the student aware of the concept of linear and nonlinear data structures.
- introduce the concept of static and dynamic memory allocation

Prerequisites

We are living in a world of data and information. Every second, a huge amount of data emerges from different sectors like business, agriculture, industry, information technology, and social media. The study of computer applications encompasses the study of organisation, flow and processing of data in various sectors using a computer. The data structure is a tool used to process digital data efficiently in terms of time and space.

Data and Information

Do you know the relationship between data and information? In computing, data is a sequence of symbols that represents an object (for example, a student, vehicle, etc.), a relationship, or an idea. Data represented using binary numbers zero and one is called digital data. Processed data is called information. So, by processing digital data, we get digital information. Modern computers process digital data to get digital information.

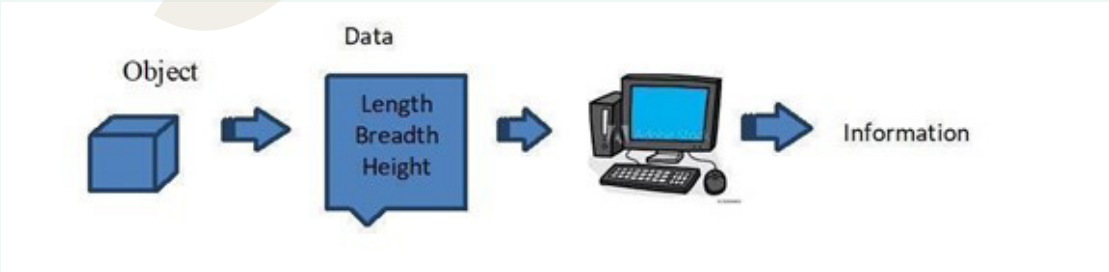


Fig 1.1.1 Data and Information

Figure 1.1.1 illustrates the relationship between data and information with an example. Here, we can see a cube, an object and data derived from the cube's length, breadth and height. From this data, we can compute the volume, the surface area, etc., the different information regarding the data.

Data types

Data type is a concept that defines an internal representation of data in memory. Every programming language has its own set of data types. The basic data type of programming language is called primitive data type. Integer, character and float/real are examples of primitive data types. Data types formed using primitive data types are called derived data types. Pointers, structure and union are examples of derived data types.

Abstract data types

In computer science, an abstract data type (ADT) is a mathematical model for data types. A set of possible values and operations on data defines it. In the example specified above, the set <length, breadth, height > is an ADT, and from this particular ADT, we can compute the volume of the cube where volume is the product of three parameters: length, breadth and height.

Keywords

Array, Queue, Stack, Linkedlist, Tree, Graph

Discussion

1.1.1 Data structure

We learned that an algorithm is a sequence of steps that a program or any computational procedure has to take. A program, on the other hand, is an implementation of an algorithm, and it could be in any programming language. During execution, programs take different inputs or data. Data structure is the way we organise the data so that it can be used effectively by the program. Thus, we can define data structure as the organisation of data needed to solve the problem. Array, Stacks, Queue, Lists and Graphs are examples of data structures that are widely used for the design and analysis of step-by-step processing of data.

1.1.1.1 Data structure in everyday life

- **Stack**

We can visualise a stack like a pile of plates placed one on top of the other. Each plate below the topmost plate cannot be directly accessed until the plates above are removed.

A stack is a linear data structure which follows the Last-In-First-Out (LIFO) principle.

Plates can be added and removed from the top only. Each plate is an element, and the pile is the stack. In programming terms, each plate is a variable, and the pile is a data structure.

Fig 1.1.2 Visualization of the stack as a pile of plates

♦ Queue

A Queue is also a linear data structure in which the elements are arranged based on the FIFO (First In, First Out) rule. It is like the passengers standing in a queue to board a bus. The person who first gets into the queue is the one who first gets on the bus. The new passengers can join the queue from the back, whereas passengers get on the bus from the front as shown in the figure 1.1.3.

A queue is a linear data structure which follows the First-In-First-Out (FIFO) principle.

Figu 1.1.3 Visualization of Queue

♦ Graph

A Graph is a network of interconnected items. Each item is known as a node, and the connection between them is known as the edge. You probably use social media like Facebook, LinkedIn, Instagram, and so on. Social media is a great example of graphs being used. Social media uses graphs to store information about each user. Here, every user is a node, just like in Graph. And, if one user, let's call him Jack, becomes friends with another user, Rose, then there exists an edge (connection) between Jack and Rose. Likewise, the more we are connected with people, the more nodes and edges of the graph keep on increasing.

Fig 1.1.4 Visualization of graph

Similarly, Google Maps is another example where Graphs are used. In the case of Google Maps, every location is considered as nodes, and roads between locations are considered as edges. When one has to move from one location to another, Google Maps uses various Graph-based algorithms to find the shortest path. Visualization of sample graph is shown in the figure 1.1.4.

1.1.2 Need for Data Structures

As applications are getting more complex and the amount of data is increasing over time, there may arise the following problems:

Processor speed: To handle a very large amount of data, high-speed processing is required, but as the data is growing day by day to the billions of files per entity, processors may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store; if our application needs to search for a particular item, it needs to traverse 106 items every time, resulting in slowing down the search process.

Multiple requests: If thousands of users are simultaneously searching data on a web server, there is a risk that even a very large server may fail during this process.

In order to solve the above problems, data structures are used. Data is organised to form a data structure in such a way that all items are not required to be searched, and required data can be searched instantly.

1.1.3 Classification of Data Structures

Data Structures are generally classified into two classes:

- Primitive Data Structures
- Non-primitive Data Structures

1.1.3.1 Primitive Data Structures

Primitive data structures are the fundamental data types supported by programming languages. They are created without the support of other data structures as a support or tool.

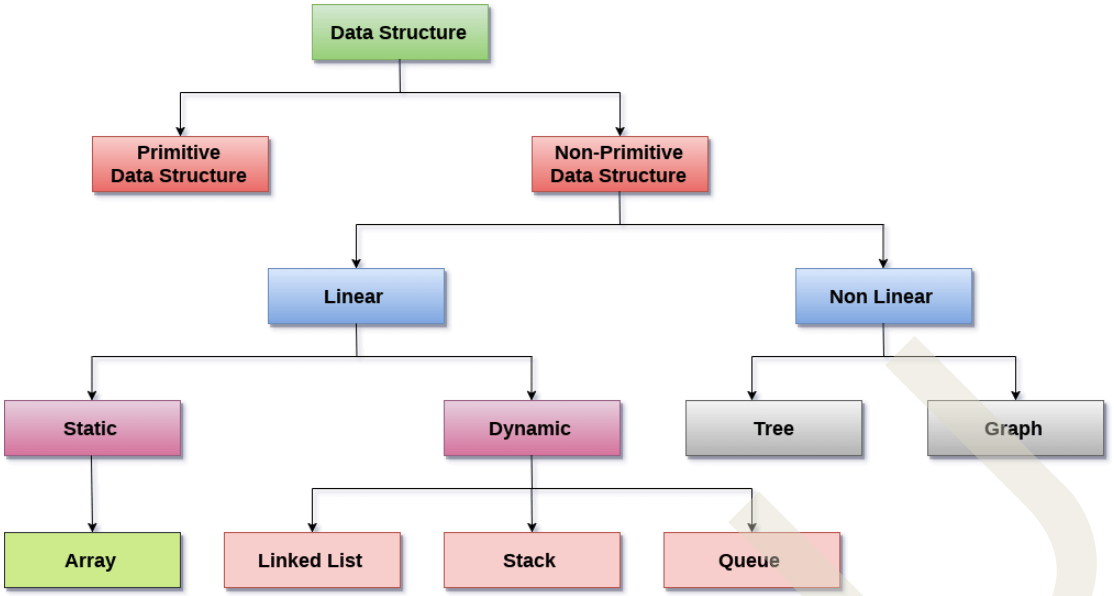


Fig 1.1.5 Classification of data structure

A primitive data structure is a basic type of data structure that stores data of a single type. The primitive data structure consists of fundamental data types like float, character, integer, etc.

Examples:

Integer :1,2,3,-1,-4, …

Real(float): 2, 0.234, √3, …

Characters: a, b, A, B, C, …

Booleans : 0 and 1

1.1.3.2 Non-Primitive Data Structures

Non-primitive Data Structures are created using primitive data structures. Non-primitive data structures are more complicated data structures and are derived from primitive data structures. Users can design these Data Structures. Examples: Lists, Graphs, Stacks and Trees.

A non-primitive data structure is considered a user-defined structure that allows storing values of different data types within one entity.

Non-Primitive Data Structures can further be classified into two categories:

- Linear Data Structures and
- Non-Linear Data Structures

1.1.4 Linear Data Structure

A data structure is called linear if all of its elements are arranged in a sequential order. In linear data structures, the elements are stored in a non-hierarchical way where each element has successors and predecessors except for the first and last elements. It is further divided into two:

- Static
- Dynamic

1.1.4.1 Static Data Structure

A static data structure is an organisation or collection of data in memory that is fixed in size. This results in the maximum size needing to be known in advance, as memory cannot be reallocated at a later point. Arrays are a prominent example of a static data structure.

Array

An array is a collection of similar types of data items, and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double. The index represents position of an element in the array. Index value of first element in the array is zero. The size of an array refers to total number of elements that an array can hold as shown in the figure 1.1.6.

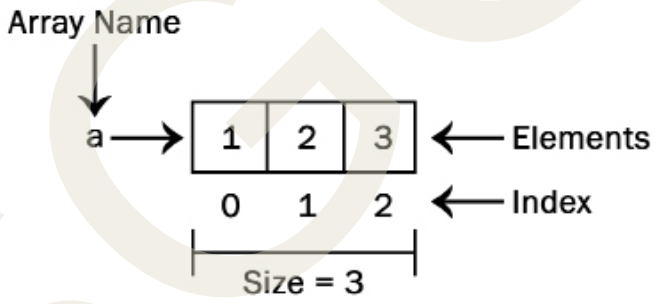


Fig 1.1.6 Visualization of the basic terminology of array

1.1.4.2 Dynamic Data Structure

In Dynamic data structure, the size of the structure is not fixed and can be modified during its operations. Dynamic data structures are designed to facilitate change of data structures during run time.

Examples:

- **Linked List**

A linked list is a linear data structure which is used to maintain a list in the memory. It can be viewed as a collection of nodes stored at different memory locations that are

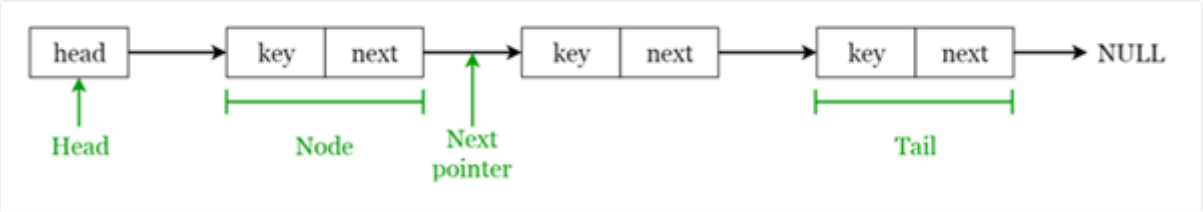


Fig 1.1.7 Visualisation of basic terminology of linked list

not contiguous. Each node of the list contains a pointer to its adjacent node, as shown in Figure 1.1.7.

♦ Stack

A stack is a linear data structure in which insertion and deletions are allowed only at one end, which is called the top. It follows the Last-In-First-Out (LIFO) methodology for

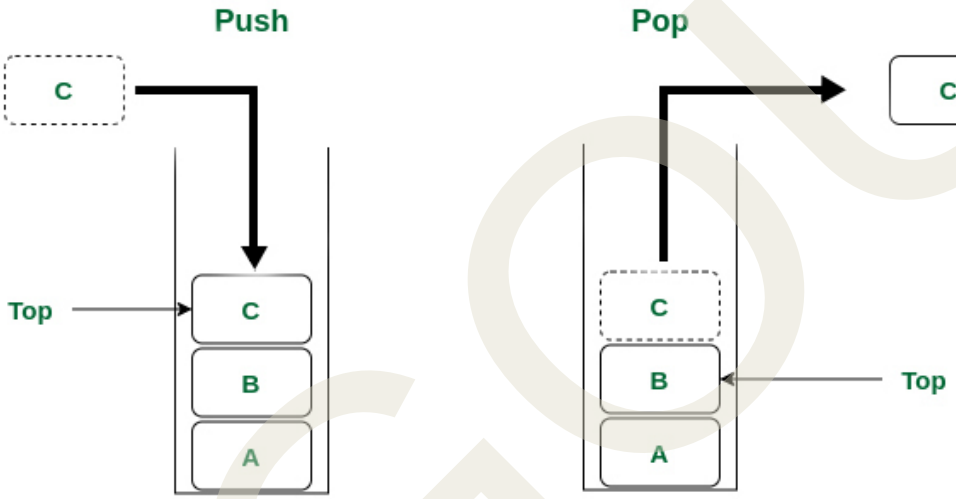


Fig 1.1.8 Stack

storing the data items. PUSH() and POP() are the two important operations related to stack. PUSH() operation is used to insert new data items into the stack, and POP() operation is used to remove or delete a data item from the stack, as shown in Figure 1.1.8.

♦ Queue

The queue is a linear data structure in which elements can be inserted only at one end, called the rear, and deleted only at the other end, called the front. It is an abstract data

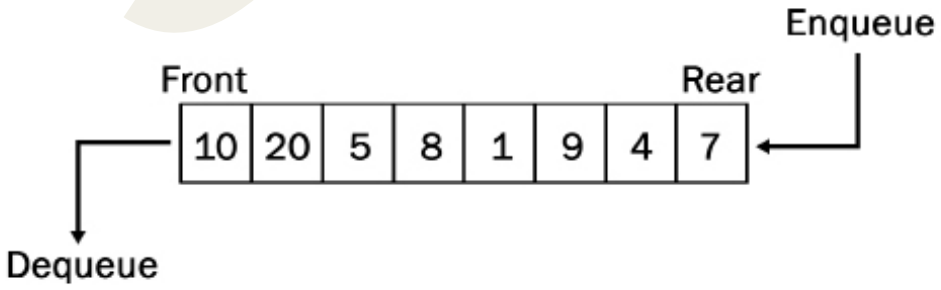


Fig 1.1.9 Queue

structure similar to a stack. The queue is opened at both ends; therefore, it follows the First-In-First-Out (FIFO) methodology for storing the data items, as shown in Figure 1.1.9. Data insertion operation in a queue is known as enqueue, and data deletion operation in a queue is known as dequeue.

1.1.5 Non-Linear Data Structures

A non-linear data structure does not form a sequence, i.e., each item or element is connected to two or more other items in a non-linear arrangement. The data elements

The files and folders in Windows Explorer are organised in a tree format.

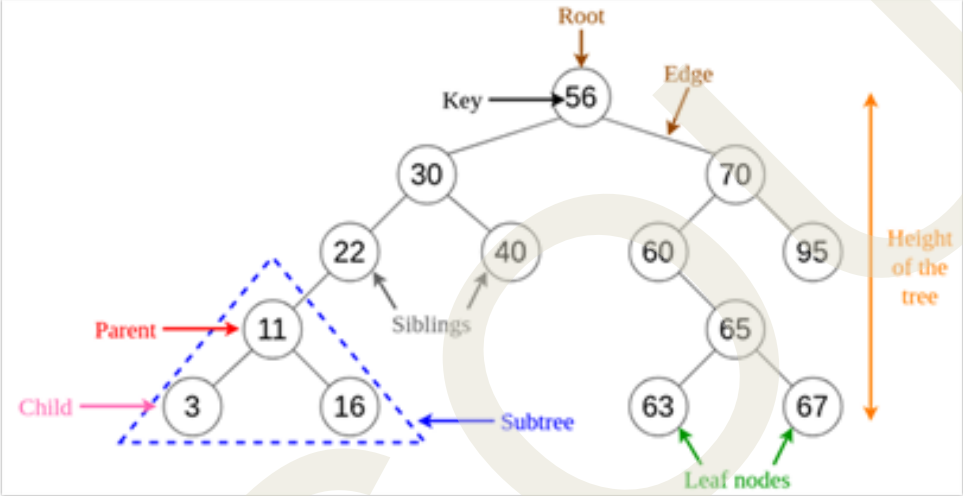


Fig 1.1.10 Visualization of the basic terminology of trees

are not arranged in a sequential structure. Examples of Non-Linear Data Structures are given below:

♦ Trees

Trees are multilevel data structures with a hierarchical relationship among their elements known as nodes. The bottommost nodes in the hierarchy are called leaf nodes,

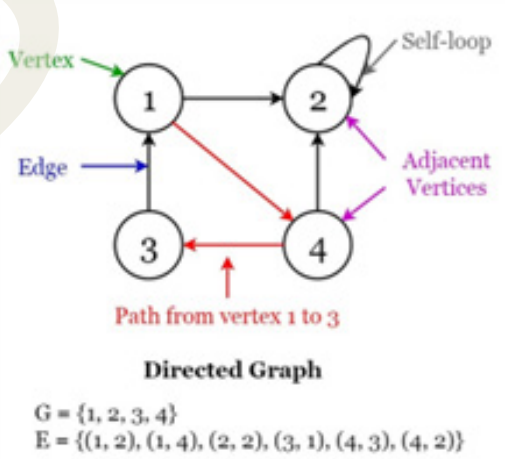


Fig 1.1.11 (a) Visualization of the basic terminology of Directed graph

while the topmost node is called root node. Each node contains pointers to point adjacent nodes. The tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes, whereas each node can have at most one parent except the root node, as shown in Figure 1.1.10.

♦ Graphs

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from a tree in the sense that a graph can have cycles, while a tree cannot have cycles.

Graphs may be directed or undirected, as shown in Figure 1.1.11(a) and Figure 1.1.11(b). A graph is said to be a directed graph if all its edges have a direction indicating the start vertex and the end vertex. Observe figure 1.1.11(a); it is a directed graph with edge set E and vertex set G. We say that the edge (1, 2) is incident from or leaves vertex 1 and is incident to or enters vertex 2 as shown in the figure 1.1.11(a). Edges from a vertex to itself are called Self-loops. In Figure 1.1.11 (a), in vertex 2, there is a self-loop.

A graph is said to be undirected if all its edges have no direction, as shown in Figure 1.1.11(b). It can traverse in both directions between the two vertices. If a vertex is not connected to any other vertex in the graph, it is called an isolated vertex, as shown in Figure 1.1.11(b).

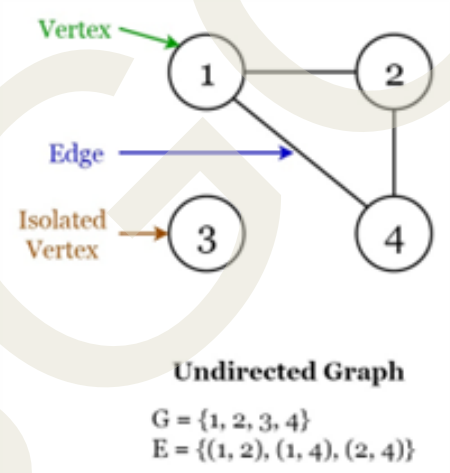


Fig 1.1.11(b) Visualization of the basic terminology of an undirected graph

A graph is defined as a pair $G = (V, E)$, where, V is a set of elements called vertices and E is a set of pairs of vertices called edges.

1.1.6 Contiguous and Non-Contiguous Data Structures

In every program, the data you handle has a specific structure or organisation. Regardless of how complex your data structures are, they can be classified into two fundamental types:

- Contiguous data structure
- Non-contiguous data structures

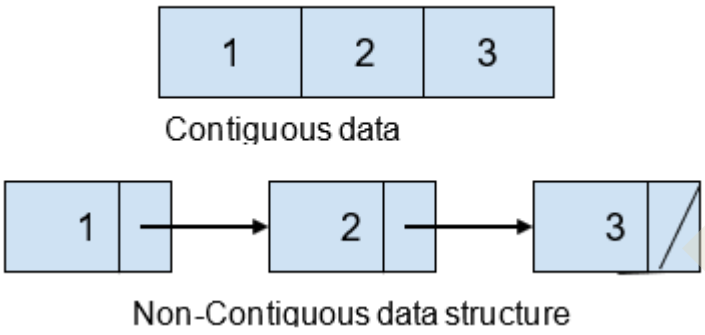


Figure 1.1.12 Contiguous and Non-contiguous data structure

1.1.6.1 Contiguous Data Structures

In contiguous data structures, data elements are stored sequentially in memory, whether in RAM or a file. An array is an example of a contiguous structure, as each element within the array is positioned adjacent to one or two other aspects. A sample of contiguous data structure is shown in Figure 1.1.12(a) and Figure 1.1.12(b).

1.1.6.2 Non-Contiguous Data Structures

In a non-contiguous structure, data are scattered in memory but linked to each other in some way. A linked list is an example of a non-contiguous data structure. A sample of non-contiguous data structure is shown in Figure 1.1.12(b).

1.1.7 Static and Dynamic memory allocation

1.1.7.1 Static Memory Allocation

In static memory allocation, the compiler allocates static memory for the declared variables. The address can be obtained using the address-of operator and assigned to a pointer. The memory is allocated during compile time. In static allocation, the compiler determines the storage size, which remains fixed and allows the compiler to easily track the addresses of these data objects in activation records at a later stage.

Features:

- In the static memory allocation, variables get allocated permanently.
- Static Memory Allocation is done before program execution.
- It uses a stack for managing the static allocation of memory
- It is less efficient
- In Static Memory Allocation, there is no memory re-usability
- In static memory allocation, once the memory is allocated, the memory size cannot change.

- In this memory allocation scheme, the unused memory cannot be reused.
- In this memory allocation scheme, execution is faster than dynamic memory allocation.
- In this allocation scheme, memory is allocated at compile time.
- In this allocation scheme, allocated memory remains reserved from the beginning to the end of the program.
- This static memory allocation is generally used for static data structures like arrays.

1.1.7.2 Dynamic Memory Allocation

Dynamic memory allocation refers to memory allocation performed during program execution (runtime). In C programming, the functions calloc() and malloc() support dynamic memory allocation. In dynamic memory allocation, memory space called the heap is allocated using these functions. The important C functions that are used for dynamic allocation of memory are discussed below.

- **malloc()**

The "malloc" or "memory allocation" function in C is used to dynamically allocate a single large block of memory with the specified size.

- **calloc()**

The "calloc" or "contiguous allocation" function in C dynamically allocates a specified number of memory blocks of a specified type. It initialises each block with a default value '0'.

- **free()**

The "free" function in C is used to deallocate or release memory that was previously allocated dynamically. The memory allocated using functions malloc() and calloc() does not automatically get deallocated. Therefore, the free() method is used whenever dynamic memory allocation occurs. It helps to minimise memory wastage by releasing it.

- **realloc()**

The "realloc" or "re-allocation" function in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory allocated previously using malloc() or calloc() is insufficient, realloc() can be used to dynamically re-allocate or adjust memory allocation. Re-allocation of memory maintains the already present value, and new blocks will be initialised with default garbage value.

Features:

- In dynamic memory allocation, variables are allocated only when your program unit becomes active.

- Dynamic memory allocation occurs during the execution of a program.
- It utilises the heap to manage dynamic memory allocation.
- It is more efficient.
- Dynamic memory allocation allows for memory reusability and the ability to free memory when it is no longer needed.
- In dynamic memory allocation, the size of allocated memory can be changed.
- This enables memory reuse. Users can allocate additional memory as needed and release it if it is no longer required.
- In this memory allocation scheme, execution is slower than static memory allocation.
- This memory is allocated at run time.
- This allocated memory can be released at any time during the program execution.
- Dynamic memory allocation is generally used for linked lists.

Recap

- The Organization of data that is needed to solve a problem is called a data structure.
- The Data Structures are generally classified into two classes:
  - Primitive Data Structures and
  - Non-primitive Data Structures.
- Fundamental data types which are supported by programming languages are called primitive data structures.
- Non-primitive Data Structures are created using primitive data structures.
- Non-Primitive Data Structures can further be classified into two categories:
  - Linear Data Structures and
  - Non-Linear Data Structures.
- A data structure is called linear if all of its elements are arranged in a linear order.
- In the nonlinear data structure, the data elements are not arranged in a sequential structure. Example: Trees, Graphs
- Linear data structure is classified into two:
  - static data structure.

- dynamic data structure.

- A static data structure is an organisation or collection of data in memory that is fixed in size. Example: Array
- In dynamic data structure, the size of the structure is not fixed and can be modified during the operations performed on it. Example: Stack, Queue, Linked list

Objective Type Questions

1. What is the term used to describe the organisation of data so that the program can effectively utilise it?
2. What is the term used to define the step-by-step procedure to solve a problem?
3. What is the term used to refer to processed data called?
4. What is the term used to describe fundamental data structures which are supported by programming languages?
5. Give the name of the data structures that are created using primitive data structures.
6. Which data structure of the elements is stored in a non-hierarchical way where each element has successors and predecessors except the first and last element?
7. What is the term used for the data structure that follows a fixed-sized organisation?
8. Give an example of static data structure.
9. Write an example of dynamic data structure.
10. Give the name of data structures that facilitate change of organisation at run time.
11. Name the data structure in which each item or element is connected with two or more other items.
12. Write an example of a non-linear data structure.
13. Write an example of a contiguous data structure.
14. Write an example for Non-contiguous data structures.

Answers to Objective Type Questions

1. Data structure
2. Algorithm
3. Information
4. Primitive data structures
5. Non-primitive data structure
6. Linear data structures
7. Static Data Structure
8. Array
9. Linked list
10. Dynamic Data Structure
11. Nonlinear Data Structure
12. Trees
13. Array
14. Linked list

Assignments

1. What is a data structure?
2. What is the use of data structure? Explain
3. What are primitive data structures? Give example
4. What are non-primitive data structures?
5. Explain the classification of non-primitive data structure in detail.
6. Distinguish between static data structure and dynamic data structure.
7. Explain the classification of linear data structure.
8. What are graphs? How does it differ from a tree?
9. What is a linked list?

10. What is a tree data structure?

11. Distinguish between static and dynamic memory allocation

12. What is Queue data structure?

Suggested Reading

1. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. "Data structures and algorithms". Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.

2. Kruse, Robert, and C. L. Tondo. "Data structures and program design in C". Pearson Education India, 2007.

3. Narasimha Karumanchi, Narasimha Karumanchi. "Data Structures And Algorithms Made Easy." (2017).

4. https://sonucgn.wordpress.com/wp-content/uploads/2018/01/data-structures-by-d-samantha.pdf

UNIT 2 Array as a Data Structure

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- explain the definition and attributes of Array
- introduce the concept of a one-dimensional array
- familiarise the concept of declaring and initialising arrays in C.
- introduce various operations on one dimensional array
- make them understand the concepts and operations of a two-dimensional array

Prerequisites

In the first unit, we defined Data structure. It is the way we need to organise the data so that it can be used effectively by the program to improve the efficiency of the program; the efficiency, in terms of time and space used for processing the data in a computer program to get information.

Data Structures are broadly classified into two: primitive data structure and non-primitive data structure. Primitive data structures are fundamental data structures supported by programming languages. On the other hand, data structures created using primitive data structures are called non-primitive data structures. The non-primitive data structures are further classified into linear and non-linear data structures. In this unit, we will discuss one of the linear data structures, Arrays.

Consider an egg tray used to store eggs. That is, suppose five eggs are placed in a single row in an egg tray. Here, we can say that the egg tray is an array because, in an array, we can store similar things. We never use the egg tray to store anything other than eggs. In the same way, only the same items can be stored in the array. We can store eggs in many ways in the egg tray. That is, if we keep eggs in only one row in the egg tray, then we call it as one-dimensional (1D) array, as shown in Figure 1.2.1. An array is a linear data structure which is used to store the same type of data.

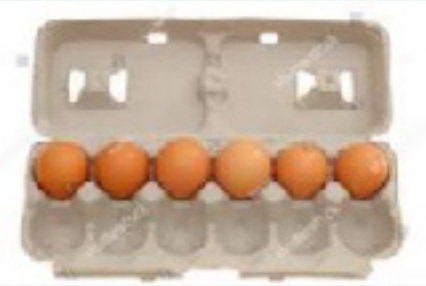


Fig 1.2.1 One-dimensional arrangements of eggs

Similarly, we can arrange eggs in two rows in the tray as linearly shown in Figure 1.2.2, which is similar to the 2D array concept. A two-dimensional array, also known as a matrix or a 2D array, is a data structure that consists of a collection of elements arranged in rows and columns.

Fig 1.2.2 Two-dimensional arrangements of eggs

Another example to show a 2D array is a chessboard. A chess board contains 64 1x1 square boxes. Each square box has a row value and column value.

Keywords

Array Name, Array Size, Array type, One-dimensional array, Two-dimensional array, multi-dimensional array

Discussion

1.2.1 Array as a data structure

In data structure, arrays are preferred to store data of similar types using a common name. Arrays are very useful for list and table processing.

In Figure 1.2.3, you can see an array of English alphabets stored in computer memory. The first data in the array is 'U', whose index is 0, and it is stored in memory location 200. Similarly, all data stored in the array have an index. Array elements are accessed

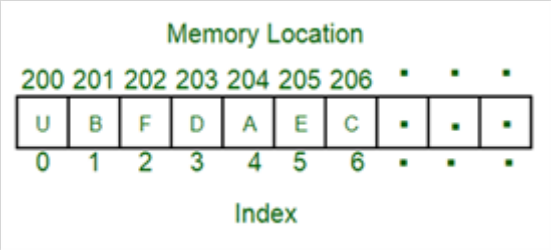


Fig 1.2.3 Array as a data structure

by using their respective index values. An array with 'n' number of elements is referred to using an index that ranges from 0 to n-1. The lowest index is called the lower bound of the array, and the highest index is called the upper bound of the array. All the elements in an array must be stored in consecutive memory locations. So, we can say an array is contiguous in nature.

An array primarily has three attributes. They are used to declare an array.

- Array Name
- Array Size
- Array type

An array is a linear data structure that gathers elements of the same data type, storing them in contiguous and adjacent memory locations.

1.2.1.1 Array Name

An array is referred to by its name. It is a user-defined variable that is used to access array elements by specifying index values.

1.2.1.2 Array Size

Array size specifies the maximum number of elements that an array can contain.

1.2.1.3 Array type

Array Type specifies the data type of the elements in an array. For example, suppose arr is an array of 10 integers. Then the array name is arr, the array size is 10, and the array type is integer since it is an array of integers.

1.2.2 Array Declaration

Declaration of the array is always language-specific. The syntax for the declaration of arrays in C language is given below.

Syntax:

Data_type name of the array [n1][n2]…[nn];

Where data_type specifies the data type of elements in the array and $[n_1][n_2]\ldots[n_n]$ are dimensions of the array. An array can be 1D,2D or nD. For example,

int ***arr*** [n1];

declare a 1D array where n1 is size of the array

int ***arr*** [n1][n2] ;

is a 2D array where n1 and n2 specify the size of the array.

For example, consider a one-dimensional array of integers with size 5. It can be declared as follows:

int ***arr***[5];

When we declare an array using the above statement, the compiler will allocate a contiguous block of memory capable of storing five integer values. For example, if 4 bytes is the size required to store an integer in a memory location, then 5*4 byte =20 bytes are needed to store the array.

1.2.3 One-Dimensional Array

A one-dimensional array is a sequential or linear data structure that stores elements of identical data types in consecutive memory locations.

1.2.3.1 Declaration of 1D array

The syntax to declare a 1D array is given below.

Syntax:

Data_type name of the array [number of elements];

Data_type specifies the data type of elements in the array, and the number of elements specifies the size of the array, which is always a positive integer. For example, the array **StudList** and **Mark** can be declared as follows

- char StudList[5];
- int Mark[5];

1.2.3.2 Initializing 1D array

Once an array is declared, we need to insert data values into it. It is called the initialisation of an array. There are two methods for initialising an array. They are discussed below.

Method 1: Compile time initialisation

In this method, the size of the array is specified along the name of the array inside a square bracket. All the data elements are specified inside a curly bracket by listing the elements using comma-separated values. If the data values are characters, then they are put inside a single inverted comma. For example, consider the initialisation of arrays **StudList** and **Mark**.

- StudList[5] = {‘ A’, ‘B’, ‘C’, ‘D’, ‘E’ };
- Mark[5]={75,80,65,85,70};

Method 2: Run time initialisation

Using runtime initialisation, users can get a chance to accept or enter different values during different runs of a program. It is also used to initialise large arrays or an array with user-specified values. In C programming language, an array is initialised at run-time using the scanf() function, as shown below.

```
//array declaration
        char StudList[5];
//array initialisation
for (i=0;i<5;i++)
{
scanf(“%c” , &StudList[i]);
}
```

Now, see how we can initialise an integer array in the same manner. It can be done as shown below.

```
// array declaration
        int Mark[5];
// array initialisation
        int Mark[5];
for (i=0;i<5;i++)
{
        scanf(“%d” , &Mark [i]);
}
```

During initialisation, if the number of elements is less than the length of the array, the remaining locations of the array are filled with the value ‘0’.

Specifying the size of an array using *macros* is considered to be an excellent practice. It is done as shown below

```
#define n=10——(This is a macro)
int Mark [n];
```

1.2.3.3 Accessing Elements from 1D Array

To access an array element, write:

Syntax:

array_name [index];

For example, let us see how to access elements of the array 'Marks' shown in Figure 1.2.4. To access the first element, we write Marks[0]; to access the second element, we write Marks[1] and so on. One thing you always remember is that indexes always start at '0' and end at 'array_size-1'.

1.2.3.4 Memory representation of a 1D array

Single-dimensional arrays are always allocated contiguous blocks of memory. This implies that all the elements in an array are always stored next to each other. For example, let us see how the array of marks is stored in memory. The schematic representation is shown in Figure 1.2.4.

Let the memory be byte-addressable; that is, one address location can hold one byte of data. For simplicity, assume the array address starts at memory location 100, and it is called the base address of the array. In C programming, we learned that data of type 'int' needs 4 bytes to be stored in memory. So, as shown in Figure 1.2.4, memory locations 100 to 103 to store 75, 104 to 107 to store 80, 108 to 111 to store 65, 112 to 115 to store 85 and 116 to 119 to store 70. Thus, the array Marks occupies 4×5=20 contiguous bytes in memory, and these bytes are reserved in the memory at the compile-time. Knowing the base address of the array, the address of the memory location of each element can be calculated easily using the following equation.

The memory address of the element, array[i] = Base address + i × (size of the data type of element in the array)

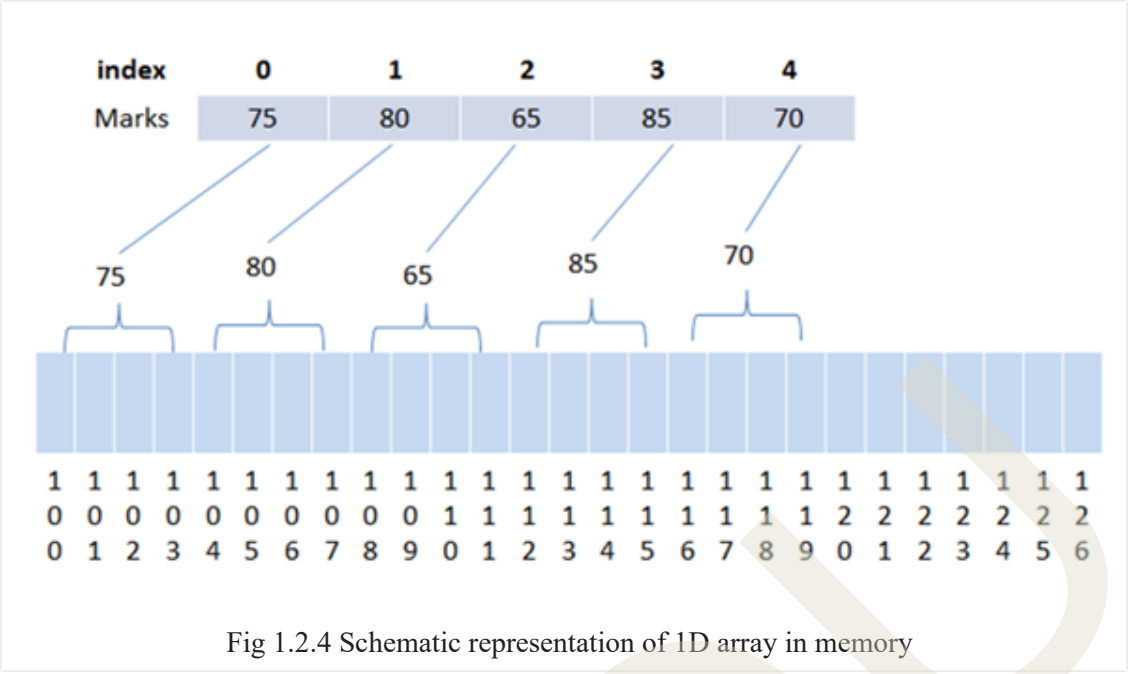


Fig 1.2.4 Schematic representation of 1D array in memory

Where i is the index of the element whose address wants to be calculated.

In the above example, can you find the address of the 2nd element? Let us see how it is calculated using the specified equation. Here, the index or value of i is '2', and the base address is 100. Also, as mentioned above, the size of the data type is four as it is an integer. So we compute Marks[2] as

Mark[2] =100 + 2 × 4

=100 + 8

=108

1.2.4 Two-Dimensional Array

We learned that a one-dimensional array is useful for representing linear lists. Suppose we want to represent a table or a matrix; in that case, an array of arrays is used to store rows and columns. We can visualise it as a table, as shown in the figure. 1.2.5. As you can see, there are two rows and three columns in the array A. The index values for the row are 0 to 1, and the index values for the column are 0 to 2, clearly shown in Figure 1.2.5.

	0	1	2
0	1	2	3
1	4	5	6

Fig 1.2.5 Two-dimensional array A [2, 3]

1.2.4.1 Declaration of 2D Array using C

We learned what a two-dimensional array is and how it can be visualised. Here, we will learn how it can be declared using C. The syntax for the declaration of 2D arrays using C programming language is given below.

Syntax:

Data_type array_name[max_rows][max_columns];

Max_rows is a variable that specifies the number of row elements, and max_columns specify the number of column elements in an array.

1.2.4.2 Initialization of 2D Array

There are two ways in which a Two-Dimensional array can be initialised.

- **Method 1: Compile time initialisation**

```
int A[2][3] = {1,2 ,3,4 , 5, 6 }
```

The above array has two rows and three columns. The elements in the braces from left to right are stored in the table and from left to right. The elements will be filled in the array in the following order: the first three elements from the left in the first row, the next three elements in the second row, and so on. Another method of compile time initialisation is given below.

```
int A[2][3] = {{1,2,3}, {4,5,6}};
```

This type of initialisation makes use of nested braces. Each set of inner braces represents one row. In the above example, there are a total of two rows, so there are two sets of inner braces.

- **Method 2: Run time initialisation**

In a one-dimensional array, at the time of runtime initialisation, we used one “for loop”. In two-dimensional arrays, we use two for loops, as shown below.

```
int x[2][3];
printf(“Enter the elements:”);
for (i=0;i<2;i++)
{
for(j=0;j<3;j++)
{
scanf(%d, &x[i][j]);
}
}
```

1.2.4.3 Representation of 2D array in memory

When it comes to mapping a two-dimensional array, most of us might wonder why this mapping is required. However, 2-D arrays exist from the user's point of view. 2D arrays are created to implement a relational database table that looks like a data structure in computer memory; the storage technique for 2D arrays is similar to that of a one-dimensional array.

The size of a two-dimensional array is equal to the product of the number of rows and the number of columns present in the array. We need to map two-dimensional arrays to the one-dimensional array in order to store them in memory.

Suppose we have a list of students: Nina, John, Alpha, Tony and Smith. We need to list the names of the students alphabetically.

Alpha	John	Nina	Smith	Tony

Fig 1.2.6 Array representation of student names

How can we arrange this in computer memory? One method we have already studied is using arrays, as shown in Figure 1.2.3. However, arrays have some limitations and disadvantages. Recall the limitations of the array.

- The number of elements to be stored in an array should be known in advance.
- An array is a static structure (which means the array is of fixed size). Once declared, the size of the array cannot be modified. The memory allocated to it cannot be increased or decreased.
- Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations, and the shifting operation is costly.
- Allocating more memory than the requirement leads to wastage of memory space, and less allocation of memory also leads to problems.

The data structure called a linked list overcomes these limitations. So, let us define a

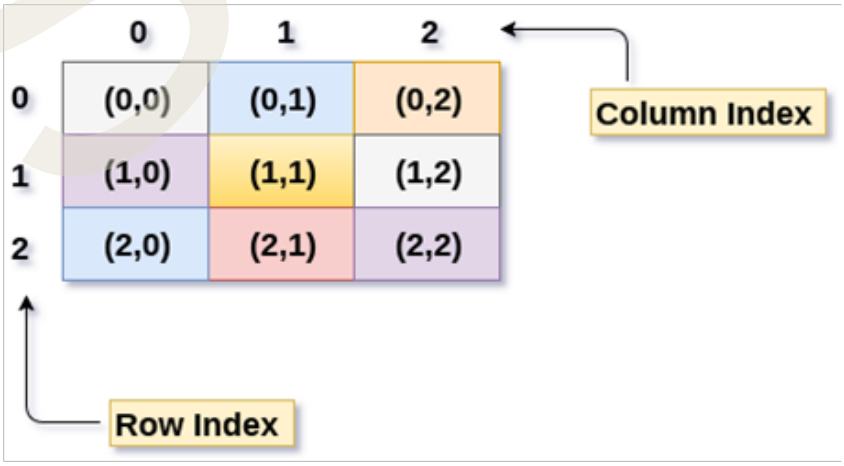


Fig 1.2.7 Visualization of 2D array

linked list. Linked List is a data structure used to store similar types of data in memory. It is a linear collection of data elements called nodes. Elements in a linked list are not stored in adjacent memory locations as in an array. They follow non-contiguous memory allocation.

A 3 X 3 two-dimensional array is shown in the given figure 1.2.7. However, this array needs to be mapped to a one-dimensional array in order to be stored in memory.

There are two main techniques for storing 2D array elements in memory. They are row-major ordering and column-major ordering.

1. Row Major ordering

In row-major ordering, all the rows of the 2D array are stored in the memory contiguously. Considering the array shown in the above image, its memory allocation according to row-major order is shown as follows.

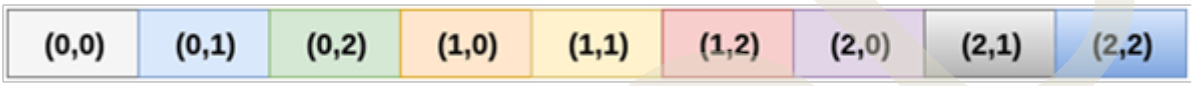


Fig 1.2.8 Memory allocation according to row-major order

First, the 1st row of the array is stored in the memory completely, and then the 2nd row of the array is stored in the memory completely, and so on till the last row, as marked in Figure 1.2.8.

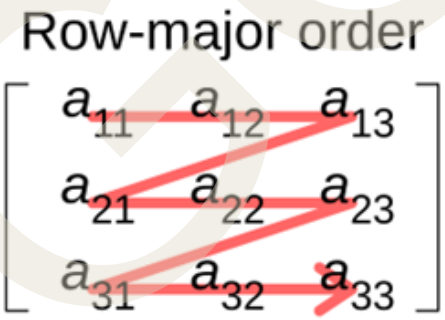
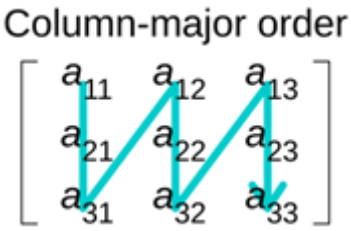


Fig1.2.9 Row major ordering

2. Column Major ordering

According to the column-major ordering, all the columns of the 2D array are stored in the memory contiguously. The memory allocation of the array, which is shown in the image, is given as follows.

a. Memory allocation according to column-major order



b. Illustration of column-major ordering

Fig 1.2.10 Column major ordering

First, the 1st column of the array is stored in the memory completely, then the 2nd row of the array is stored in the memory completely, and so on till the last column of the array, as shown in figure 1.2.10.

As in the 1D array, knowing the base address, we can compute the address of any of the elements in the 2D array. But it is necessary that the column size 'n' is given. For row-major ordering, the address of the element at position (i, j) can be computed as follows

Address of a[i][j] = B+((i*n)+j)*size

where B is the base address, n is the column size, and i and j are indexes of the element whose address wants to be computed.

In a column-major implementation with the starting index of '0', the address of any element is calculated using the equation given below.

Address of a[i][j]=B+((j*m)+i)*size

where m is the number of rows and B is the base address of the given 2D array.

1.2.4.4 Accessing of elements in 2D arrays

How to access 2D array elements? Using row index and column index, we can access elements of a 2D array. For example, we can access elements in the first row and second column of the array given below using A[0][1].

	0	1	2
0	1	2	3
1	4	5	6

Similarly, we can access all elements in a 2D array as

A[0][0]=1, A[0][2]=3 and so on.

1.2.4.5 Printing 2D array elements

You recall that 1D array elements can be printed using a single for loop. But to print a 2D array, we use two nested for loops, as shown in the program code below.

```
int A[2][3] = {{1,2,3},{4,5,6}};
for (i=0; i<2; i++)
{
        for (j=0;j<3;j++)
        {
                printf("%d", A[i][j]);
        }
}
```

1.2.5 Advantages and disadvantages of array

1.2.5.1 Advantages of array

In an array, using the index number, an element can be accessed very easily.

The search process can be applied to an array easily.

2D Array is used to represent matrices.

For any reason a user wishes to store multiple values of similar type, then the Array can be used and utilised efficiently

1.2.5.2 Disadvantages of array

The array is static, which means its size is always fixed. The memory allocated to it cannot be increased or decreased.

The array is homogeneous, i.e., only one type of value can be stored in the array.

1.2.6 Static and Dynamic Memory Allocation

1.2.6.1 Static Memory Allocation

In static array allocation, memory is allocated at compile time with a fixed array size. In this type of memory allocation, we can't change, alter or modify the size of the array.

Example:

Consider an array initialized as int a[5]={1,2,3,4,5};

It will create an array named 'a', and the size of the array is 5. we can only insert five

elements into this array. Adding a sixth element is not possible in this array because the size of the array is fixed.

1.2.6.2 Dynamic Memory Allocation

In dynamic memory allocation, memory is allocated at run time but does not have a fixed size. Suppose we want to create an array of any random size, and then we can use dynamic memory allocation. Consider an array shown in Figure 1.2.11 below.

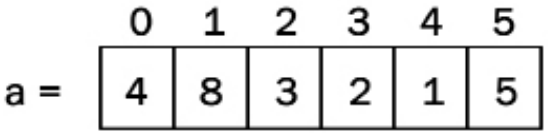


Fig 1.2.11 Array of 6 elements

In this array, the size of the array is seven, and seven elements fill out all the index positions. Suppose we want to enter three more elements in this array. In this case, we need three more index positions to store additional elements. So, the size of the array needs to be changed from 7 to 10. This procedure of changing the array size during the run time of program execution is known as dynamic memory allocation. C programming language provides four library functions to support dynamic memory allocation. They are malloc(), calloc(), free(), and realloc().

Recap

- An Array is a linear data structure that contains a collection of data of the same type referred to by a common name.
- All the elements stored in the array have an index. The elements of the array are referred to using the index.
- An array with an 'n' number of elements is referred to using an index that ranges from 0 to n-1
- The lowest index of an array is called the lower bound of the variety, and the highest index is called the upper bound of the array.
- In computers, all the elements in an array are stored in consecutive memory locations.
- Different attributes of an array are Array name, Array type and Array size
- The syntax for declaration of 2D array using C is

Data_type array_name[max_rows][max_columns];

- We need to map a two-dimensional array to a one-dimensional array in order to store them in the memory.
- There are two main techniques for storing 2D array elements in memory: Row major ordering and column-major ordering.
- In row-major ordering, all the rows of the 2D array are stored in the memory contiguously.
- In column-major ordering, all the columns of the 2D array are stored in the memory contiguously.
- Memory allocation for an array is of two types:
    1. Static memory allocation
    2. Dynamic memory allocation
- In static array allocation, memory is allocated at compile time with a fixed array size.
- In dynamic memory allocation, memory is allocated at run time but does not have a fixed size.

Objective Type Questions

1. Name the data structure commonly used for list and table processing.
2. What type of data structure is Array?
3. What is the maximum index that could be used to refer to an array with an 'n' number of elements referred to using an index?
4. What order are the in-memory array elements stored?
5. How do the elements of the array are referred to?
6. What is the lower bound of an array?
7. What is the common name used for the linear data structure that contains a collection of data of the same type referred to?
8. A program P reads 500 integers in the range [0..100] representing the scores of 500 students. It then prints the frequency of each score above 50. What would be the best way for P to store the frequencies?

    a. An array of 50 numbers

b. An array of 100 numbers

c. An array of 500 numbers

d. A dynamically allocated array of 550 numbers

9. What is the output of the following piece of code?

```
#include<stdio.h>
int main()
{
        int mark[5] = {19, 10, 8, 17, 9};
        printf("%d", mark[1]);
        printf("%d", mark[4]);
}
```

10. The declaration int a[2][5] will allocate _____ bytes

11. Which of the following correctly declares an array?

    int arr[20];

    int arr;

    Arr{20};

    array arr[20];

12. In what type of order are all the rows of a 2D array stored contiguously in memory?

13. What type of ordering are all the columns of the 2D array stored in the memory contiguously?

Answers to Objective Type Questions

1. Array
2. Linear
3. n-1
4. consecutive

5. index
6. lowest index value
7. Array
8. (A)
9. 10 and 9
10. 40 bytes
11. (A)
12. Row major
13. Column major

Assignments

1. How to represent an array as a data structure?
2. How do you represent a 2D array in a data structure?
3. Explain Row Major ordering and Column major ordering.
4. Explain the advantages and disadvantages of arrays.
5. Explain static and dynamic memory allocation.
6. Explain the compile-time and run-time initialisation of the array

Suggested Reading

1. Sharma, A. K. "Data Structures using C", 2e. Pearson Education India, 2013.
2. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. "Data structures and algorithms". Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.
3. Weiss, Mark Allen. "Data structures and algorithm analysis". Benjamin-Cummings Publishing Co., Inc., 1995.
4. https://sonucgn.wordpress.com/wp-content/uploads/2018/01/data-structures-by-d-samantha.pdf

UNIT 3 Stack and Queue

Learning Outcomes

After the successful completion of the unit course, the learner will be able to:

- explain the need for the stack as a Last In First Out data structure.
- introduce different operations on the stack.
- familiarise infix, prefix and postfix notations.
- explain the need for the queue as a First In First Out data structure
- familiarise different types and operations of queues
- narrate applications of the queue

Prerequisites

In units 1 and 2, we learned the basics of data structure and discussed array data structure in detail. In this unit, we have to discuss two other important data structures: stack and queue.

Imagine that you have a collection of coins, and you want to arrange them on a table. Suppose you are going to place each of the coins on top of the other coin, as shown in Figure 1.3.1. We can add any number of coins on the top of the other coins. Similarly we can remove coin from the top of the coin. This type of arrangement of data is called a stack. Here, the last inserted coin from the top position will be removed first, which follows the Last-In-First-Out (LIFO) principle to access elements from the stack of coins. In computing applications, a stack works in the same manner. That is a stack is a linear data structure in which elements of the same type are placed one above the other. In the stack, items are inserted and deleted from the top. To access elements of a data structure, we have to use some operations. Push() and Pop() are the two important operations used for inserting and deleting elements in a stack. In Unit 1, we have already discussed that a stack follows the LIFO(Last-In-First-Out) Principle.

Fig 1.3.1 Stack of coins

Imagine another real-life situation where people are waiting in a queue in front of an ATM. If a new person enters the queue, then he will be added to the back side of the queue, and if a person leaves the queue, then the person at the front of the queue will be removed, as shown in Figure 1.3.2

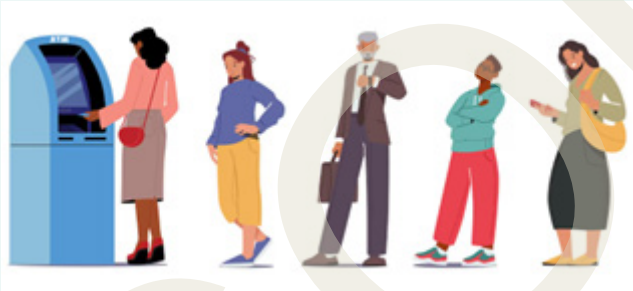


Fig 1.3.2 ATM queue

In computing a queue, a data structure follows the same concepts explained above. That is, a queue is a linear data structure where insertion take place at the back side of the queue and deletion from the front end of the queue. That is first inserted value is the first to be deleted. Unlike stack, a queue data structure has two ends: FRONT end and REAR end. The elements are inserted into the queue at the REAR end, and elements are deleted from the queue at the FRONT end. In Unit 1, we have already discussed that a queue follows the FIFO(First-In-First-Out) Principle. Operating systems mainly use the concepts of queues to manage processes and resources.

Key Concepts

LIFO, Push, pop, peek, Polish notations, Infix, prefix, postfix, FIFO, Enqueue, Dequeue

Discussion

1.3.1 Introduction to Stack

This unit explores one of the foremost crucial linear data structure stacks. The stack can be defined as an abstract data type, a concrete and valuable tool for real-world problem-solving. On the other hand, the stack may be a linear data structure arrangement where all the insertion and deletion are done at the same end. A stack is often implemented by using both arrays and linked lists.

A stack is a simple data structure used for storing data (like linked lists). In a stack, the order in which the data arrives is essential. A real-world example of a stack is the plate arrangements at a marriage party.

Fig 1.3.3 Stack Operation, Plates Arrangements

Figure 1.3.3 shows the example of a stack operation, i.e., the plate arrangements in a party. At a marriage party, the plates are arranged one above the other after cleaning. The cleaned plates are placed on top of the arrangements, followed by one another. When a plate is required, it is taken from the top of the arrangement. If you add a new plate to the stack, the previous topmost plate becomes inaccessible. You can access it only after removing the newly added plate from the top of the stack. If you remove all the items from a stack, you can access them in reverse chronological order. The recently added item will be the first item you remove from the stack, and the last item will be the first added item to the stack.

A stack is an ordered list in which insertion and deletion are done at one end, called the 'top'. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO). We can implement a stack in any programming language, such as C, C++, Java, Python, or C#.

A stack is a collection of elements where items can only be added or removed at one end, known as the top of the stack. Stacks are also referred to as LIFO (last in, first out) structures.

1.3.1.1 Basic Stack Structure

A stack can be represented in memory using different methods. A sample stack struc-

ture is shown in Figure 1.3.4.

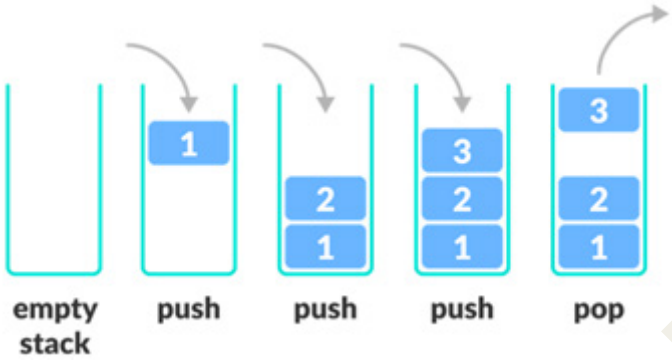


Fig 1.3.4 Visualization of the stack

1.3.1.2 Stack Representation

There are two other important ways to represent a stack: Using a one-dimensional array and a single linked list. Representation of stack is discussed in the following two sections.

a. Array Representation of stack

To represent a stack using an array, First, we need to allocate a memory block large enough to hold the stack's full capacity. Then, starting from the first location in the memory block, we can store the stack items sequentially. In Figure 1.3.5, The nth item in the stack is represented by item n, while l and u represent the index range of the array being used. Usually, the values of these indices are one and SIZE, respectively. The top is a pointer indicating the position in the array up to which the stack items are filled. Using this representation, the following two parameters can be stated as:

Empty: $TOP < l$

Full: $TOP \geq u$

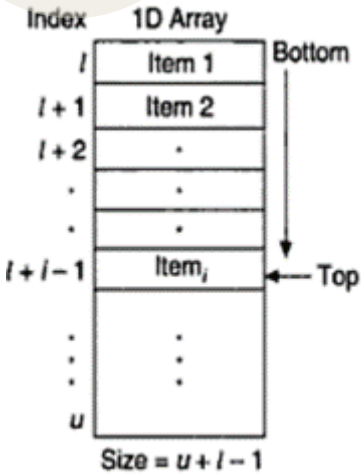


Fig 1.3.5 Array representation of the stack

b. Linked List Representation of stack

While representing stacks with arrays is easy and convenient, it only supports a fixed-sized stack. In many applications, the size of the stack can change during program execution. A clear solution to this problem is to represent a stack using a linked list. A single linked list structure is enough to represent any stack. Here, the DATA field stores the ITEM, while the LINK field points to the next item. Figure 1.3.6 illustrates a stack represented using a linked list. In the linked list representation, the first node in the list corresponds to the current item at the top of the stack, and the last node contains the bottom-most item. Therefore, a PUSH operation adds a new node to the front, and a POP operation removes a node from the front of the list.

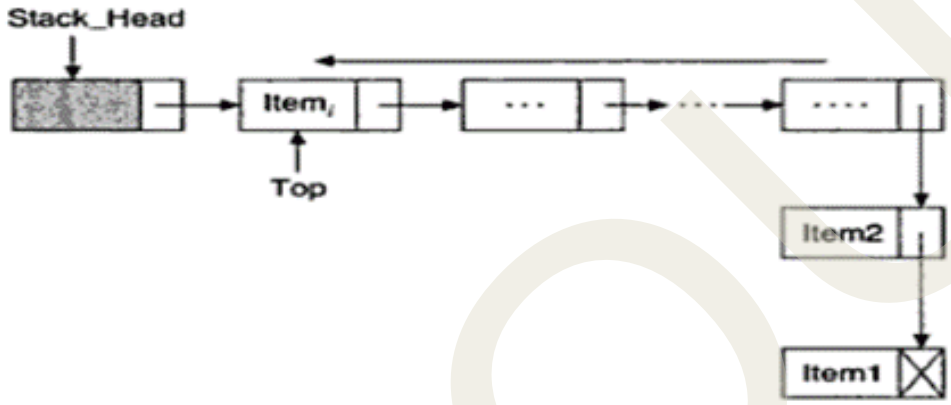


Fig 1.3.6 Linked list representation of the stack

1.3.1.3 Basic Operations

The stack is a linear data structure that follows a particular order in which the operations are performed; the order may be LIFO (Last In, First Out). The most basic stack operations are push and pop. They are described below.

- Push(): The push operator is used to add an item at the top of the stack. If the stack is full, then the stack is said to be in an overflow state.
- Pop(): pop operator is used to delete an element from the top of the stack. If the stack is empty, then the stack is said to be in an underflow state.

Basic operations associated with stacks are:

1. PUSH():- Used to insert an element into the stack
2. POP():- Used to delete an element from the stack

To calculate the stack efficiently, we need to check the status of stack operations. Therefore, the following functions are added to the stack operation.

- Peek(): The peek operator is used to get the top data item of the stack without removing it, and the peek operator displays the topmost item in the stack.

- isFull(): This operator is used to check whether the stack is full or not.
- isEmpty(): This operator is used to check whether the stack is empty or not.

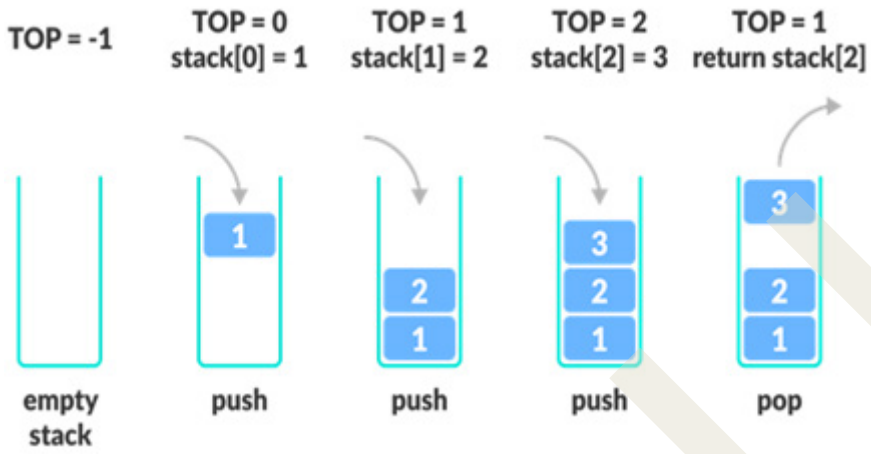


Fig 1.3.7 Working of stack operations

Figure 1.3.7 illustrates the operational sequence of stack operations. The top functions as a stack pointer, managing items within the stack. To initialise the stack, set TOP to -1. To check if the stack is empty, verify if TOP = -1. When the stack is empty, pushing an item increments the value of the TOP pointer and places the new item at the position indicated by TOP. This cycle continues as TOP takes values 0, 1, and 2 in succession. Ultimately, three items are pushed onto the stack, resulting in a stack containing three items. In PUSH operations, it is essential to check the stack is already full before pushing items from the stack. The function isFull() is used for it.

When an item is removed from the stack, the stack performs a POP(return stack) operation. Popping an item, we return the item pointed to the TOP pointer and decrement the stack pointer value (value=2). As a result, the most recently added data item, 'Item3', popped out from the stack. Thus, we can remove all the added data elements from the stack through pop operations. For POP operation, check the status of the stack to see if it is already empty or not. The function isEmpty() is used for checking.

1.3.1.3.1 Push Operation

The push operation inserts a new data item into the stack or adds an item to the stack. The steps of the Push operation are given below,

Step 1: Check if the stack is full using the function isFull()

Step 2: If the stack is full, it produces an error message and exit.

Step 3: If the stack is not full, increment the TOP pointer to a point next to space.

Step 4: Add the data element to the stack location where the TOP pointer is pointing.

Step 5: Returns success.

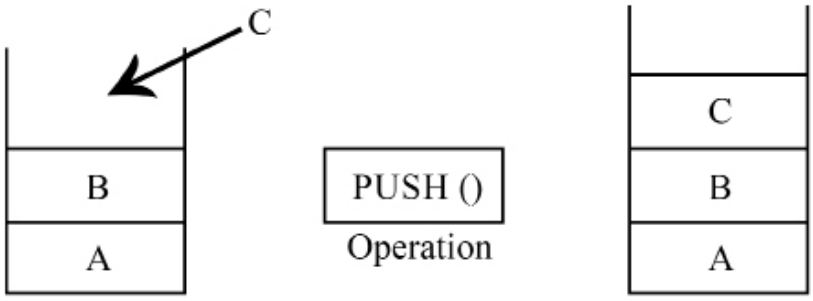


Fig 1.3.8 Working of push () operation

1.3.1.3.2 Pop Operation

Removing or returning the last inserted element from the stack is known as the POP operation. The pop() operation removes the data element and deallocates the memory space. A POP operation involves the following steps.

Step 1: Check if the stack is empty using the function is Empty().

Step 2: If the stack is empty, it produces an error message and exits.

Step 3: If the stack is not empty, access the data element at which the TOP pointer is pointing.

Step 4: Decrease the value of the TOP pointer by 1.

Step 5: Returns success.

Fig 1.3.9 (a) Working of pop () operation

1.3.1.3.3 Peek Operation

The peek operator evaluates the efficiency of the stack operations. This operator is printing the topmost element from the stack. For example, the stack given below in Fig 1.3.9(b), peek() function returns the value of C, because C is the element in place of top.

1.3.1.4 Applications of Stack

There are a number of applications of the stacks. The four major ones are discussed briefly below. The compiler internally uses the stack when we execute any recursive function. The stack is also used to evaluate a mathematical expression and to check the parentheses in an expression. The following are some of the applications in which

stacks play an important role.

- Undo/Redo operation
- Arithmetic Expression Evaluation
- Expression Conversion
- Syntax Parsing
- Backtracking
- String Reversal
- Tree traversal

1.3.1.5 Evaluation of Arithmetic Expression

An arithmetic expression consists of operands and operators. Operands are variables or constants, and operators are of various types, such as:

- Addition (+), subtraction (-), Multiplication (*), division (/), Exponentiation (^), modulo (%).
- Relational operators (<, >, <=, >= etc)
- Boolean operator (AND, OR, NOT, XOR, etc)

A simple arithmetic expression is given below:

A+B*C-E^F

The problem with evaluating the given expression is the order of evaluation.

Solution: We can assign precedence and associativity to each operator. For example, a set of usual operators with their precedence and associativity is given in Table 1.3.1. The precedence value 1 represents the highest precedence, and the value 5 represents the lowest precedence in the given table.

Table 1.3.1 Precedence and associativity of operators

Precedence	Operator	Description	Associativity
1	() []	Parentheses Square bracket	Left-to-Right
2	^	Exponentiation	Right-to-Left
3	*, /, %	Multiplication, division, modulus	Left-to-Right
4	+/-	Addition, subtraction	Left-to-Right
5	<, <=, >, >=	Relational operators	Left-to-Right

We can evaluate the expression A + B * C - E ^ F as:

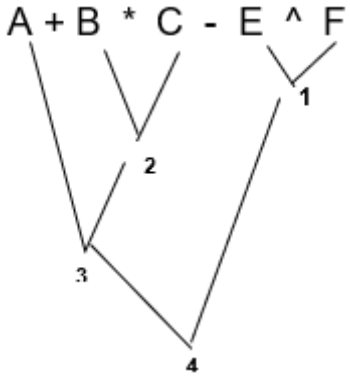


Fig 1.3.10 Evaluation of Arithmetic Expression

The above expression will be executed in the sequence of 1,2,3,4

It should be noted that the above rules for precedence and associativity vary from one programming language to another. Also, the above method of expression evaluation is inefficient because repeated scanning is required. This problem can be solved by using the following steps:

1. Conversion of a given expression into special notations.
2. Evaluation of an object code using stack: Place the lowest precedence operator first then next, lowest precedence operator so on, so that the operator with the highest precedence is at the top of the stack.

1.3.1.6 Polish Notation

The process of writing the operators of an expression either before their operand or after the operand or in between the operands is called the polish notation. There are three notations to represent an arithmetic expression.

- Infix notation
- Prefix notation
- Postfix notation

1.3.1.6.1 Infix notation

Infix notation or expressions are a common type of mathematical notation. In this type of notation, the operator is placed between its operands. For example, consider A and B as two operands that contain values 2 and 3. Then, the expression A+B is an infix expression, where the operator "+" is placed between the operands A and B. We can also write the same expression as 2+3. This notation is called infix because the operator comes in between the operands. In Infix notation, values in stack are stored like this, operand, operator, operand

Syntax:

<operand> <operator> <operand>.

2. Prefix notation

Prefix notation defines that an operator should be present as a prefix before operands. This notation is also known as polish notation. For example, suppose A+B is an expression where a and b are operands, and + is the operator. Then, the prefix expression of this expression is +AB. This notation is called a prefix because the operator comes before operands.

Syntax:

<operator><operand><operand>

In prefix notation the values in stack are placed in order that operator, operand, operand.

3. Postfix notation

Postfix notation defines that the operator should be present as a suffix after operands. This notation is also known as reverse polish notation. For example, suppose A+B is an expression where a and b are operands, and + is the operator. Then, the postfix expression of this expression is AB+. This notation is called postfix because the operator comes after operands. In postfix notation, values placed in stack in the order of operand, operand, operator.

Syntax:

<operand><operand><operator>

Some additional examples of infix, prefix and postfix expressions are shown below:

Infix Notation	Prefix Notation	Postfix Notation
A+B	+AB	AB+
(A-C)*B	*-ACB	AC-B*
A+(B*C)	+A*BC	ABC*+

1.3.1.7 Conversion of infix to postfix expression

An infix expression is an expression in which the operator is in between the operands. For example, a + b / c - d is an infix expression. We can convert the infix expression into a postfix expression using the following rules:

Step 1: Scan the expression from left to right.

Step 2: Whenever we come across an operand in the expression, we push it onto the stack.

Step 3: If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.

Step 4: If the incoming symbol is '(', push it onto the stack.

Step 5: If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.

Step 6: If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

Step 7: If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then, test the incoming operator against the new top of the stack.

Step 8: If the incoming operator has the same precedence as the top of the stack, then use the associativity rules.

- If the associativity is from left to right, then pop and print the top of the stack, then push the incoming operator.
- If the associativity is from right to left, then push the incoming operator.

Example: Convert the infix notation into postfix notation

The infix notation is A+B*C/(E-F)

Move	Input string	Output stack	Output
1	A+B*C/(E-F)		A
2	A+B*C/(E-F)	+	A
3	A+B*C/(E-F)	+	AB
4	A+B*C/(E-F)	+*	AB
5	A+B*C/(E-F)	+*	ABC
6	A+B*C/(E-F)	+/	ABC*
7	A+B*C/(E-F)	+/(	ABC*
8	A+B*C/(E-F)	+/ (	ABC*E
9	A+B*C/(E-F)	+/(-	ABC*E
10	A+B*C/(E-F)	+/(-	ABC*EF
11	A+B*C/(E-F)		ABC*EF-
12	A+B*C/(E-F)	+/	ABC*EF-/+

The result is ABC*EF-/+

1.3.1.8 Conversion of Infix to Prefix Expression

An infix expression is characterised by a mathematical expression where the operators are placed between operands. Let us consider an arithmetic expression a + b / c - d. Calculating the value of this expression, first, evaluate the value of "b/c" and add the operand "a" to the evaluated value. The operand "d" is subtracted from the calculated result, and this value is the final result of this expression. This is the mathematical solution for the "a + b / c - d" arithmetic expression. The infix expression can be converted to both prefix and postfix notations using stack operations. An expression "a + b" is an infix notation; it is in the format of "a" is an operand, then "+" is an operator and finally 'b' is also an operand. This infix notation can be converted into prefix notation as "+ a b". First, the operator "+", then the operand "a" and finally the next operand "b".

The steps to convert infix expression to prefix expression are given below.

Step 1: First, reverse the given infix expression.

Step 2: Scan the characters one by one.

Step 3: If the character is an operand, copy it to the prefix notation output.

Step 4: If the character is a closing parenthesis, then push it to the stack.

Step 5: If the character is an opening parenthesis, pop the elements in the stack until we find the corresponding closing parenthesis.

Step6: If the character scanned is an operator

- If the operator has precedence greater than or equal to the top of the stack, push the operator to the stack.
- If the operator has precedence lesser than the top of the stack, pop the operator and output it to the prefix notation output and then recheck the above condition with the new top of the stack.

Step 7: After all the characters are scanned, reverse the prefix notation output.

Example: Convert the infix expression into prefix notation.

The infix notation is (P+(Q*R)/(S-T))

Move	Symbol Scanned	Stack	Output
1	)	)	-
2	)	))	-
3	T	))	T
4	-	))-	T
5	S	))-	TS
6	(	)	TS-

7	/	)/	TS-
8	)	)/)	TS-
9	R	)/)	TS-R
10	*	)/)*	TS-R
11	Q	)/)*	TS-RQ
12	(	)/	TS-RQ*
13	+	)+	TS-RQ*/
14	P	)+	TS-RQ*/P
15	(	Empty	TS-RQ*/P+

Reverse the obtained expression TS-RQ*/P+.

The final result is that the prefix expression is + P/*QR-ST.

Some of the additional examples of converting infix to postfix notations and infix to prefix notations are given below. Understand each of these examples and do these conversions on your own.

Example of infix to postfix notation conversion:

No	Infix	Prefix
1	A+B	AB+
2	A+B-C	AB+C-
3	(A+B)*(C-D)	AB+CD-*
4	A*B/C	AB*C/
5	2+3*4	234*+
6	A*(B+C)/D-G	ABC+*D/G-

Example of infix to prefix notation conversion:

No	Infix	Prefix
1	A+B	+AB
2	A+B-C	-+ABC

3	(A+B)*(C-D)	*+AB-CD
4	A/B*C-D+E/F/(G+H)	+-*/ABCD//EF+GH
5	((A+B)*C-(D-E))*(F+G)	*-*+ABC-DE+FG
6	A-B/(C*D/E)	-A/B/*CDE

1.3.1.9 Evaluation of Postfix Expression

In an infix expression, it is difficult for the machine to keep track of and find out the operator's priority and importance in an expression. However, the postfix expression determines the operator's priority and importance. So, it is easier for a machine to execute a postfix expression than an infix expression. The postfix notation is used to represent algebraic expressions. An expression written in postfix form is evaluated faster compared to infix notation. The evaluation rule of postfix expressions is given below:

Step 1: Reading an expression from left to right, push the element into the stack if it is an operand.

Step 2: POP the two operands from the stack if the element is an operator (except NOT operator).

Step 3: NOT an operator, POP one operand from the stack and then evaluate that operand.

Step 4: PUSH back the result of the evaluation.

Step 5: Repeat it till the end of the stack.

- **Algorithm for Evaluation of Postfix Expression**

Step 1: Read the expression from left to right

Step 2: Read the first element for the first time

Step 3: If the element is an operand, then:

- PUSH the element in the stack

Step 4: If the element is an operator, then

- POP two operands from the stack
- Evaluate the expression formed by the two operands and the operator.
- PUSH the result of the expression in the stack end.

Step 5: if there are no more elements, then POP the result

Else, Go to Step 1.

For example, evaluate the postfix expression:

AB+C*D/

Where A=2, B=3, C=4, and D=5 starting from left to right.

1. The first element is an operand A, PUSH A, into the stack.

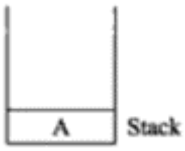


Fig 1.3.11 illustration of step 1

2 The second element is operand B, and PUSH B is also added to the stack.

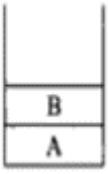


Fig 1.3.12

3. The third element, "+", is an operator, POP two elements from the stack. Then, evaluate A and B.

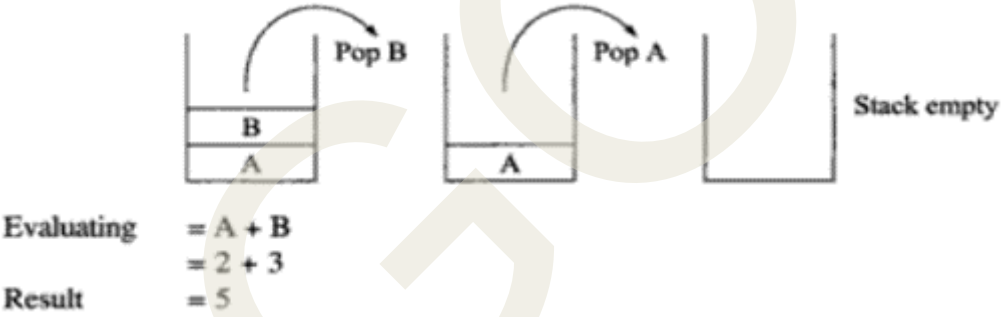


Fig 1.3.13

4. PUSH the result, element five, into the stack.

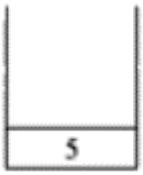


Fig 1.3.14 illustration of step 4

5. The next element, C, is an operand; PUSH it into the stack.

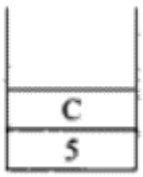


Fig 1.3.15 illustration of step 5

6. Next, "*" is an operator, POP two operands from the stack, then evaluate elements five and C.

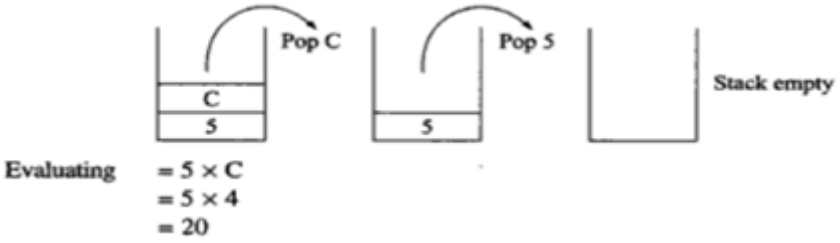


Fig 1.3.16 illustration of step 6

7. PUSH the result 20 into the stack.

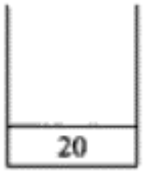


Fig 1.3.17 illustration of step 7

8. Next D is an operand; PUSH it into the stack.

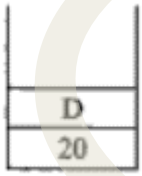


Fig 1.3.18 illustration of step 8

9. Next, "/" is an operator.

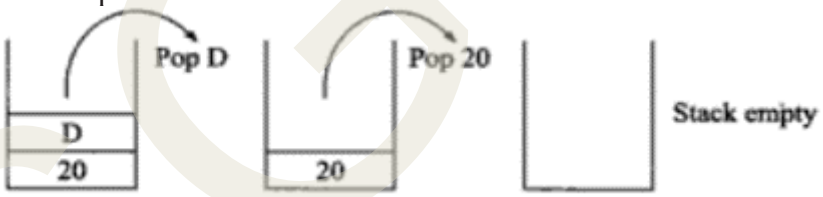


Fig 1.3.19 illustration of step 9

POP two operands from the stack and evaluate 20 and D.

Evaluating

$20/D = 20/5 = 4$

End of expression. Finally, the result is value 4.

1.3.2 Introduction to Queue

The queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The deletion end of the queue is called the front end, and the insertion end is called the rear end. The first element in a queue

will be the first to be removed from the list. Therefore, the queues are also called FIFO (First in First Out) lists. Let's look at a real-life example: A queue of people standing in line to enter a venue. A new person arriving would join the end of the line, while the person at the front would leave the line and enter the venue.

Definition: A queue is an ordered list where insertions are done at one end (rear), and deletions are done at the other end (front). The first element to be inserted in the queue is the first to be deleted from the queue. Hence, it is called the First in First out (FIFO) or Last in Last out (LILO) list.

An important example of a queue in a computer system occurs in a time-sharing system. The programs with the same priority form a queue, and each program in the queue will execute first in the first-out manner. Operating systems also use several different queues to control processes within a computer. The scheduling of program execution is typically based on a queuing algorithm that tries to execute programs as quickly as possible.

The queue data types have the following operations: first, initialise the queue and check whether the queue is empty or full. If the queue is empty, insert a new element into the queue and continue the insertion process until the queue is full. On the other hand, if the queue is full, retrieve the first element and delete the element using the First in First out (FIFO) method.

A queue is a linear data structure like a stack, but the queue is open at both ends, called FRONT and REAR ends. The basic principle of queue data structure is FIFO(First-In-First-Out).

The structure of a queue is open at both ends. One end of the queue is always used to insert data, which is called enqueue, and the other is used to remove data, which is called dequeue.

Figure 1.3.20 represents a real-world example of a queue. The Figure is a ticket window counter queue; all three people are standing in the queue in a first-in, first-out manner. The first person who enters will get the first ticket from the counter.

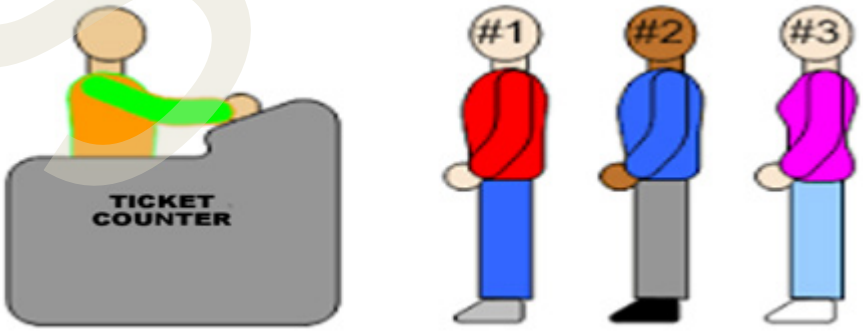


Fig 1.3.20 Ticket Window Counter Queue

The person who comes second gets the ticket second. So, the person who comes last gets the tickets last.

Fig 1.3.21 Vehicle Queue in a Single-lane One-Way Road

Figure 1.3.21 is an example of a real-world queue. The Figure represents a single-lane one-way road, where the vehicle entering the one-way road will exit first.

1.3.2.1 Different Types of Queues

The queue is an important data structure for storing and retrieving data, and hence, the queue is used extensively among all the data structures. The queue is like any queue that follows a first-in, first-out mechanism for data retrieval. The first data that gets into the queue will be the first to be taken out of the queue. The second one would be the second to be retrieved, and so on. The queue in the data structure is of the following types.

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Deque (Double-Ended Queue)

1.3.2.1.1 Simple Queue

A simple queue is the most basic queue where insertion occurs at the rear end of the queue, and deletion occurs at the front end of the queue. Simple queue-related applications are memory management, pipes, call centre phone systems, and interrupt handling.

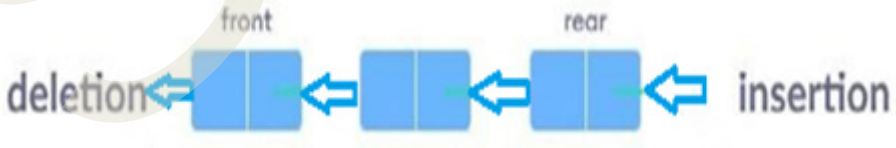


Fig 1.3.22 Simple Queue Representation

All the simple queue nodes are connected sequentially. The pointer of the first node points to the value of the second, and so on.

1.3.2.1.2 Circular Queue

A circular queue permits better memory utilisation than a simple queue when the queue

has a fixed size. In this queue, the last node points to the first node and creates a circular connection. Thus, the circular queue allows us to insert an item at the first node of the queue when the last node is full and the first node is free. The main advantage of a circular queue over a simple queue is better memory utilisation.

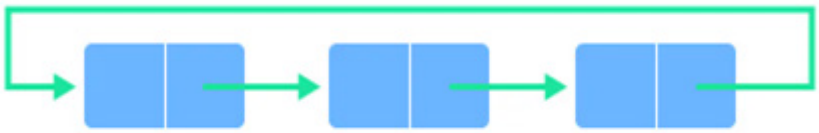


Figure 1.3.23 Circular Queue Representation

The insertion of a data element in a circular queue happens at the front end of the queue, and deletion at the rear end of the queue.

1.3.2.1.3 Priority Queue

A priority queue is a particular type of queue in which each element is associated with a priority and is served according to its priority. It makes data retrieval possible only through a predetermined priority number assigned to the data items.

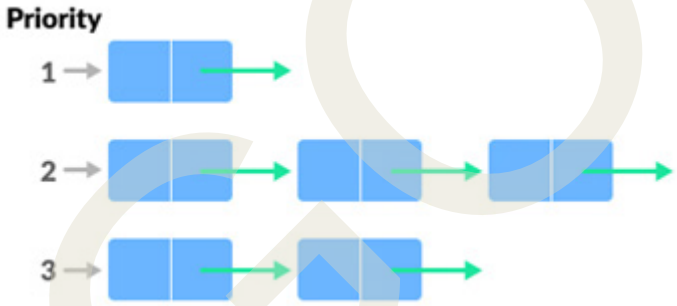


Fig 1.3.24 Priority Queue Representation

In the priority queue, elements are inserted and deleted based on the following rules. First, an element of higher priority is processed before any aspect of lower priority. Then, the second two elements with the same priority are processed according to the order they were added to the queue.

1.3.2.1.4 Double-Ended Queue

A double-ended queue is, in short, called a deque. In a double-ended queue, insertion and deletion operations can be done at both the front and rear end. Therefore, the double-ended queue does not follow the FIFO (First In, First Out) rule.

Fig 1.3.25 Double-Ended Queue Representation

1.3.2.2 Basic Operations in Queue

The queue data structure can also be implemented using arrays, linked lists, pointers and structures. For the sake of simplicity, the queue data structure can be implemented using a one-dimensional array. There are two important basic operations in the queue data structure. They are enqueue and dequeue.

1. enqueue operation

- The enqueue is the process of inserting an element into the queue. In the queue, elements are always inserted at the rear end of the queue.

2. dequeue operation

- The dequeue is the process of removing an element from the queue. Elements from the queue are always removed from the front end of the queue.

A few more functions are required to make the queue mentioned above operate efficiently. They are:

peek()

This function helps to see the data at the front end of the queue, and this function gets the element at the front end of the queue without removing it.

isFull()

This function checks if the queue data structure is full. We use a single dimension array to implement a queue; we check for the rear pointer to reach MAXSIZE to determine that the queue is full.

isEmpty()

This function checks if the queue is empty or not. It returns true if the queue contains a zero number of elements, which means the queue is empty.

1.3.2.2.1 Enqueue Operation in Queue

The enqueue operation in the queue is performed at the rear end of the queue. The data insertion process cannot be performed if the queue is full. It is checked using the function isFull(). The queues maintain two data pointers, front and rear. Therefore, queue data structure operations are comparatively more difficult to implement than a stack data structure. When the queue has space, the enqueue operation inserts an element and updates the queue 'rear' pointer by incrementing rear = rear+1.

The following steps are used to insert data elements into a queue.

Step 1: Check if the queue is full.

Step 2: If the queue is full, produce an overflow error and exit.

Step 3: If the queue is not full, increment the rear pointer to the next empty space.

Step 4: Add a data element to the queue location where the rear is pointing.

Step 5: Return success.

Figure 1.3.26 represents the enqueue operation in the queue. The queue data structure contained 44, 55 and 66 data elements in the first, second and third memory locations, respectively. The current status of a queue is Q [1] = 44, Q [2] = 55 and Q [3] = 66.

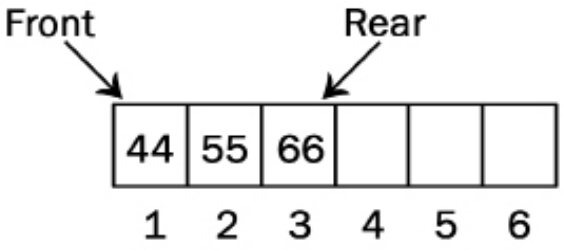


Fig 1.3.26 Operation on Queue

Suppose we are going to insert a fourth data element, 77, the value of rear is incremented by 1 and become 4 i.e, rear = rear + 1. into the queue at the rear end. Then, the queue's new status contains four data elements, and the newly added element is Q [4] = 77, which is shown in Figure 1.3.27.

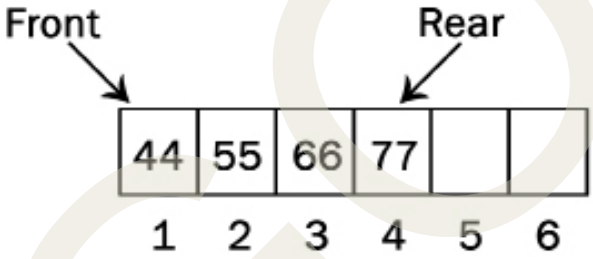


Fig 1.3.27 After Enqueue Operation on Queue

1.3.2.2.2 Dequeue Operation in queue

The dequeue operation removes elements from the front end of the queue, but the queue must not be empty for dequeue operation. It is checked using the function isEmpty(). Accessing data from the queue is a process of two tasks: first, access the data where the front end is pointing and second, remove the data after access. In the dequeue operation, elements from the front end are removed, and the 'front' pointer is updated.

The following steps are used to perform the dequeue operation:

Step 1: First, check if the queue is empty using the function isEmpty().

Step 2: If the queue is empty, produce an underflow error and exit.

Step 3: If the queue is not empty, access the data where the front is pointing.

Step 4: Increment front pointer to point to the next available data element.

Step 5: Return success.

Figure 1.3.28 represents the dequeue operation in the queue. The queue data structure

contained three data elements, 34, 15, and 54, in the first, second, and third memory locations. In the queue data elements, the dequeue operation removes elements from the front end of the queue.

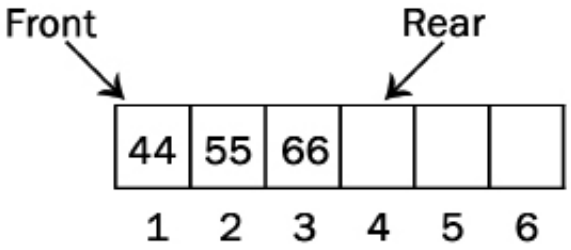


Fig 1.3.28 Dequeue Operation on Queue

The first memory location of the queue contains value 34, that is, Q [1] = 34, and the front pointer points to value 1. The element Q [1] = 34 is deleted in the queue in the first dequeue operation. After deletion of the first element, the front pointer is incremented to '1 + 1 = 2'. It focuses on the second element in the queue Q [2] =15. After the first dequeue operation, the remaining queue data elements are shown in Figure 1.3.29. Repeating the dequeue operation, all the inserted data elements are removed from the queue, and the queue will become empty.

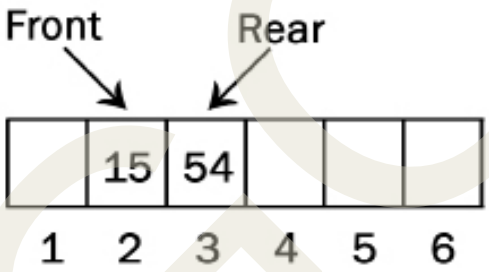


Fig 1.3.29 After Dequeue Operation on Queue

Understand the basic operations like enqueue and dequeue in the queue data structure; some supportive functions are used in the queue to improve the efficiency of the operations. The supporting functions of a queue are the peek(), isFull() and isEmpty() functions. The peek() function always returns the head of the queue. The isFull() function checks for the rear pointer to reach MAXSIZE to determine that the queue is full. If the front value is less than MIN or 0, the isEmpty() function tells that the queue has not yet been initialised.

1.3.2.3 Applications of Queue

The abstract data type queue follows the first-in, first-out principle. A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket. The following are some of the applications of queue data structure.

The first application of the queue is if you are planning a tour via Indian railways, you have to book the ticket. After completing the ticket booking process, your booked ticket is in a waiting list category, like W/L1, W/L2...etc. This means that the ticket is

in a queue of tickets waiting to be confirmed, as per the increasing order of waiting numbers. Therefore, in case of confirmed train ticket cancellations by others, your wait-listed ticket can fetch you a confirmed berth. The W/L1 numbered ticket is removed from the front end of the waiting queue, and the ticket will be confirmed.

The second application is that operating systems are required to handle multiple tasks called jobs; these jobs seek to use the processor. However, the processor can handle only one task at a time. Therefore, in a multitasking operating system, jobs are lined up (queued) and then given access to the processor according to some order. The simplest way is to provide access to the processor on a FIFO basis according to the order in which the jobs arrive with a request for the processor.

Recap

- The stack is a linear data structure that follows the LIFO (Last-In-First-Out) order in which the operations are performed.
- A stack may be represented in the memory using a one-dimensional array and a single linked list.
- The push operation refers to inserting an element in to the stack.
- The pop operation refers to removing (accessing) an element from the stack.
- The peek() is one of the stack operations that returns the value of the topmost element of the stack without deleting that element from the stack.
- A stack data structure can be implemented using a one-dimensional array.
- An arithmetic expression can be represented in three different but equivalent notations: infix, prefix, and postfix, can process using stack.
- The stack data structure can be used to evaluate and convert expressions in infix, prefix, and postfix notations.
- The process of writing the operators of an expression before the operands or after the operands or in between the operands is called the polish notation.
- The infix notation is the notation used in arithmetical and logical statements. In this notation, the operator is placed between the operands.
- In prefix notation, the operator is prefixed to operands that are written before its operands.
- In postfix notation, the operator is postfixed to the operands, and the operator is written after the operands.
- The queue is an abstract data structure that is similar to stacks.

- The queue is open at both ends. One end is always used to insert data called rear, and the other end removes data called front.
- A Queue is a FIFO (First in, First Out) data structure where the element added first will be deleted first
- An example of a queue in daily life is the people waiting in line at a bank. Man first comes out of line.
- The queue data structure is classified into four types.
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - Double Ended Queue
- There are two important basic operations in the queue data structure. They are enqueue and dequeue.

Objective Type Questions

1. Which data structure is required to check whether an expression contains balanced parenthesis?
2. What is the process of inserting an element into the stack?
3. What is the process of removing elements from the stack?
4. What error occurs when the user tries to remove an element from the empty stack?
5. Consider a stack that already has five elements and has a size of five. What error occurs when the user pushes one element in the stack?
6. To keep track of the current topmost element of the stack, we need to maintain one variable. Is this statement True/False?
7. What will be the initial value with which the top is initialised?
8. A linear linked list may represent a stack. Is the given statement true?
9. (a+b)*(c+d) is an example of which expression?
10. What is the postfix expression for this infix expression A+B*C+(D*E)

11. What is the ordering principle of the queue?

12. What are the two operations implemented on a queue?

13. In which end data can be added to the queue?

14. Which end does data delete from the queue?

15. In which queue does the insertion of an element take place at both ends, but deletion occurs at one end only?

16. Which data structure is required for breadth-first traversal on a graph?

17. If the elements "A", "B", "C", and "D" are placed in a queue and are deleted one at a time, in what order will they be removed?

Answers to Objective Type Questions

1. Stack
2. PUSH
3. POP
4. Underflow of Stack
5. Overflow
6. True
7. -1
8. True
9. An infix expression
10. ABC*+de*+
11. FIFO
12. Enqueue(insertion) and Dequeue(Deletion)
13. Rear
14. Front

15. Input-restricted/output-restricted

16. Queue

17. ABCD

Assignments

1. Transform the following expression to prefix and postfix.

   `P+Q−Z`

2. Transform the following expression from prefix to infix.

   `+−ABC`

3. What is the postfix expression for the infix expression?

   a-b-c

4. What is the postfix expression for the following infix expression?

   a/b^c-d

5. What is the corresponding postfix expression for the given infix expression?

   a*(b+c)/d

6. What are all the primitive operations in the stack?

7. In which principle does the stack work?

8. Find out some practical implementations of the stack.

9. Convert the following infix expression into a postfix expression.

   1. (A+B)*C

   2. (A+B)*C/E

   3. ((A-(B+C))*D)/(E+F)

10. Evaluate the following postfix expressions

    1. AB+C-BA+C/-

    2. ABC+*CBA-+*

    Where A=1, B=2, C=3.

11. How does FIFO describe a queue?

12. Explain different types of queue data structure.

13. Explain the types of queue operations in the data structure.

Suggested Reading

1. Sharma, A. K. "Data Structures using C", 2e. Pearson Education India, 2013.

2. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. "Data structures and Algorithms". Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.

3. Weiss, Mark Allen. "Data structures and algorithm analysis". Benjamin-Cummings Publishing Co., Inc., 1995.

4. https://sonucgn.wordpress.com/wp-content/uploads/2018/01/data-structures-by-d-samantha.pdf

UNIT 4

Circular Queue, Double Ended Queue and Priority Queue

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- explain the limitations of the linear queue
- introduce operations in circular queue
- explain how to implement a double-ended queue using a circular queue
- familiarise priority queue implementation using a circular queue

Prerequisites

In Unit 3, we discussed the basic concepts of queue data structure. This unit is a continuation of Unit 3. That is, the topic of what are the different types of queues is mainly to be studied in this unit.

Consider a real-life example of a printer. A printer is an item that we use to print various types of documents in daily life. Suppose we give more than one print job to the printer; the printer will keep all of them in a queue. Then, based on the order of requests received, each print job is generally done. Once the end of the queue is reached, processing starts from the beginning if necessary. This type of queue is called a circular queue. We can use enqueue and dequeue operations, which we learned in unit 3 as basic queue operations in a circular queue, to add and delete elements in a queue. A traffic light control system mainly uses the concept of a circular queue. That is, each coloured light will be in the queue for some seconds in circular order, as shown in Figure 1.4.1

Fig 1.4.1 Traffic signal control system

In any normal queue, we can only insert elements at the REAR end and FRONT end to remove an element from the queue. Can we add and remove data from both ends of a queue? Yes. A double-ended queue is a special type of queue. Where we can add and delete values through the two ends of the queue.

Consider another situation, such as checking the status of a call centre with a call. Not all calls in a call centre have to be of the same nature. Some customers' needs are more important than others. So, in situations like this, we have to prioritise those calls based on their priority. So, we can use a priority queue to store prioritised calls. Priority queue is another type of queue data structure.

Keywords

Queue, Circular Queue, Linear Queue and its Limitations, Operators in Circular Queue, Double Ended Queue, Priority Queue, Recursion, Palindrome Process

Discussion

1.4.1 Introduction to Queue

A queue is a linear type of data structure which follows the first-in, first-out (FIFO) order. In other words, if the FIFO principle is implemented with an array, then that will be called the queue data structure. In the queue data structure, the two most commonly used operations are enqueue() and dequeue(). The enqueue() function is used to insert elements into the queue, and the dequeue() function to delete elements from the queue. The two ends in the queue data structure are called the front and rear end of the queue.

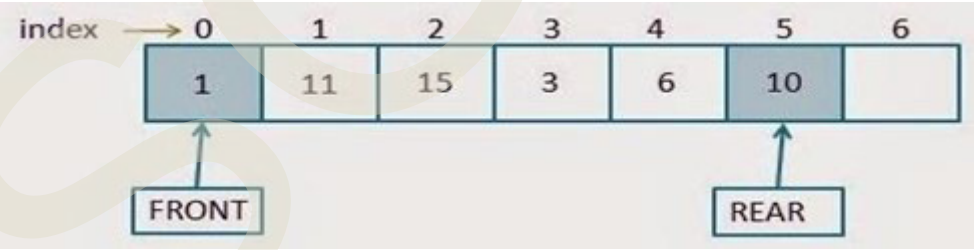


Fig 1.4.2 Front and Rear End of a Queue Data Structure

Figure 1.4.2 represents a queue data structure; it is clear from the figure that the queue has a front end and a rear end. The above queue contains 6 data elements; the value of the front end is 1, and the value of the rear end is 10.

1.4.2 Types of Queues

The queue is a linear data structure used to represent a linear list. It allows the insertion of an element to be done at one end and the deletion of a component to be performed at the other end. In a queue, the first element inserted is the first one to be deleted or

removed. Therefore, a queue follows the FIFO (First In, First Out) structure. The queue data structures are classified into four categories. They are;

1. Simple Queue
2. Circular Queue
3. Double Ended Queue (D - Queue)
4. Priority Queue

1.4.2.1 Simple queue

In a simple queue, the insertion occurs at the rear end of the queue, and deletion occurs at the front end of the queue. In a simple queue, adding an element in a queue is called an enqueue operation, and the process of removing an element from a queue is called a dequeue operation. A simple queue follows the FIFO (First in, First out) rule. The best example of a simple queue process is in the checkout line at the supermarket cash counter, where the first person in the line is usually the first to be checked out. Figure 1.4.3 is the graphical representation of a simple queue.

Fig 1.4.3 Simple Queue

The simple queue contains two pointers: the front pointer and the rear pointer. The front pointer includes the location of the front element of the queue, and the rear pointer consists of the location of the rear element of the queue. The algorithm for the insertion and deletion of data in a simple queue is given below:

The condition FRONT = -1 will indicate that Q is Empty.

The condition REAR >= N-1 will indicate that Q is Full.

Where N represents the maximum capacity or the total number of elements that the queue can hold

- **Enqueue Algorithm**

In the simple queue insertion process (enqueue algorithm), Q, N, F, R, and Item are the variables used to create this algorithm. Here, F and R are the front and rear elements of the queue. The queue 'Q' contains N data elements. The 'item' is the new item in the simple queue list at the rear end of the queue.

QINSERT (Q, N, F, R, Item)

Initially F = R = -1

Step1: [Check for Overflow Condition]

If R >= N-1 then:

write “Overflow”

return

Step2: [Increment REAR Pointer]

If F == -1, then: [Queue initially empty]

set F = 0 and R = 0

else:

set R = R + 1

[End of If Structure]

Step 3: [Insert Item]

set Q[R] = Item

Step 4: Return

- **Dequeue Algorithm**

In the simple queue deletion process (dequeue algorithm), Q, N, F, and R are the variables used in this algorithm’s creation. Here, F and R are the front and Rear elements of the queue. The queue Q consists of N elements. This procedure deletes an element from the front end of the queue.

QDELETE (Q, N, F, R)

Initially F = R = -1

Step 1: [Check for Underflow Condition]

If F == -1, then Design and Explain Prim’s Algorithm for Finding the Minimum Spanning Tree (MST) of a Graph

write “Underflow”

return (0)

Step 2: set Y = Q[F] and

[F] = NULL

Step 3: [Increment Front Pointer]

If F == R, then: [Queue has only one element left]

set F = R = -1

else:

set F = F + 1

[End of If Structure]

Step 4: Return (Y)

♦ **Example and Limitation of simple queue**

An example of a simple queue in the real world is a single-lane one-way road, where vehicles enter and exit in the order they arrive. More real-world examples can be seen as queues at the ticket windows and bus stops. If the simple queue is full, that is R == N-1, and if you delete elements from the simple queue's front end, then there are some free spaces available at the front end of the queue. This free space cannot be used to insert new elements. This is the one drawback of a simple queue.

1.4.2.2 Circular Queue

A circular queue is a queue that is implemented in a circular form. It is also a linear data structure, which follows the principle of FIFO (First In, First Out). Instead of ending at the last position, the queue starts again from the first position, making it behave like a circular data structure. Figure 1.4.4 is the Graphical representation of a circular queue.

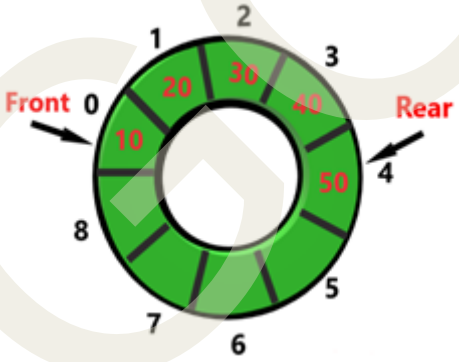


Fig 1.4.4 Circular Queue

The circular queue structure overcomes the problem of unutilised space in a simple linear queue implemented as an array. In the array implementation, the queue can be reported as full even when there are empty slots available in the queue. This is because, at the time, the rear part of the queue has reached the end of the array. When the queue gets full, there is a possibility of the vacant element at the beginning of the array if some of the elements are deleted.

Figure 1.4.5 represents a data organisational graphical image of a circular queue, the queue having six data elements; its rear value is 50, and the front value is 23. A circular queue is similar to a linear queue because it is based on the FIFO (First in, First Out) principle, except that the last position is connected to the first position. However, not all circular queue data structures are linear.

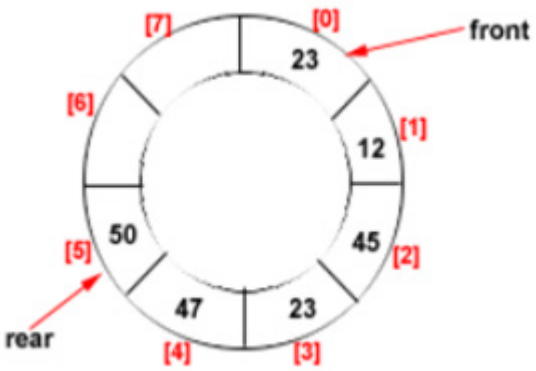


Fig 1.4.5 Circular Queue having Rear = 5 and Front = 0

The following are the operations that can be performed on a circular queue. In the circular queue, there are two pointers, front and rear. The front pointer is used to get the front(first) element from the circular queue, and the rear pointer is used to get the rear (last) element from the circular queue. Thus, the queue has insertion and deletion operations; they are enqueue and dequeue.

The enqueue operation is used to insert the new value into the circular queue. The new element is always inserted from the rear end. This operator works as follows:

Step 1: Check if the queue is full

Step 2: For inserting the first element, set the value of the front to 0

Step 3: Circularly increase the rear index value by 1

Step 4: Add the new component of the position pointed to by the rear pointer.

The dequeue operation is used to delete an element from the circular queue. The deletion in a queue always takes place from the front end. This operator works as follows;

Step 1: Check if the queue is empty

Step 2: Return the value pointed by the front pointer

Step 3: Circularly increase the front index value by 1

Step 4: For the last element, reset the values of the front pointer and the rear pointer to -1

Enqueue and dequeue operations are based on the queue data insertion and deletion algorithms in the circular queue.

The two insertion and deletion algorithms are given below:

- **The Circular Queue Data Insertion Algorithm**

In this data insertion process, CQueue, Rear, Front, N and Item are the variables used in this algorithm creation. First, data elements are inserted into the queue; CQueueis is a circular queue where data is stored. The Rear represents the location in which the data

element is to be inserted. The front represents the location from which the data element is to be removed. Finally, N is the maximum size of the queue, and the item is the new item in the circular queue list.

Insert Circular Q (CQueue, Rear, Front, N, Item)

Initially, Rear = 0 and Front = 0.

Step 1: If Front = 0 and Rear = 0, then Set Front = 1 and go to step 4.

Step 2: If Front =1 and Rear = N or Front = Rear + 1

then Print: "Circular Queue Overflow" and Return.

Step 3: If Rear = N, then Set Rear = 1 and go to step 5.

Step 4: Set Rear = Rear + 1

Step 5: Set CQueue [Rear] = Item.

Step 6: Return

♦ **The Circular Queue Data Deletion Algorithm**

After inserting all the data elements into the circular queue, the queue executes a delete operation. In this data deletion process, CQueue, Rear, Front, N and Item are the variables used in the algorithm. The CQueue is a circular queue where to store data. The rear represents the location in which the data element is to be inserted, and the front means the location from which the data element is to be removed. The front component of the queue is assigned to the item.

Delete Circular Q(CQueue, Front, Rear, Item)

Initially, Front = 1.

Step 1: If Front = 0, then

Print: "Circular Queue Underflow" and Return.

Step 2: Set Item = CQueue [Front]

Step 3: If Front = N, then Set Front = 1 and Return.

Step 4: If Front = Rear, then Set Front = 0 and Rear = 0 and Return.

Step 5: Set Front = Front + 1.

Step 6: Return.

The following graphical representation is the data insertion and deletion process in a circular queue data structure. For example, the below given circular queue has five data locations, i.e. N=5.

1. Initially, Rear = 0, Front = 0.

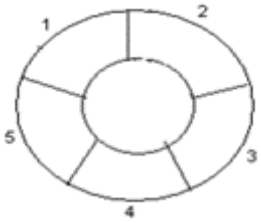


Fig 1.4.6 illustration of step1

2. Insert 10, Rear = 1, Front = 1.

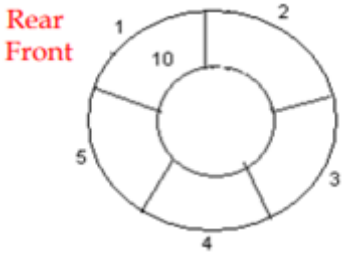


Fig 1.4.7 illustration of step 2

3. Insert 50, Rear = 2, Front = 1

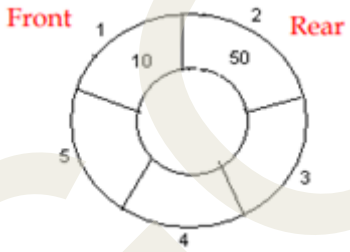


Fig 1.4.8 illustration of step 3

4. Insert 20, Rear = 3, Front = 1.

Fig 1.4.9 illustration of step 4

5. Insert 70, Rear = 4, Front = 1.

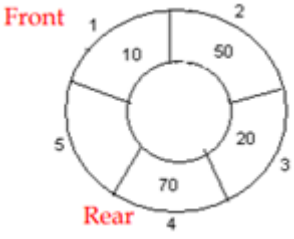


Fig 1.4.10 illustration of step 5

6. Delete Front(10), Rear = 4, Front = 2

Fig 1.4.11 illustration of step 6

7. Insert 100, Rear = 5, Front = 2.

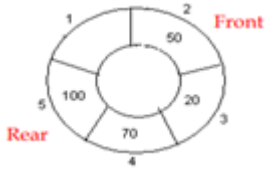


Figure 1.4.12 illustration of step7

8. Insert 40, Rear = 1, Front = 2.

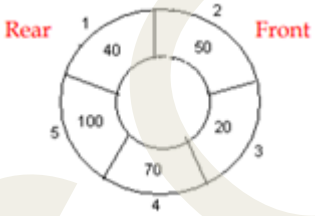


Fig 1.4.13 illustration of step 8

9. Insert 40, Rear = 1, Front = 2.

As Front = Rear + 1, so Queue overflow.

Fig 1.4.14 illustration of step 9

10. Delete Front(50), Rear = 1, Front = 3

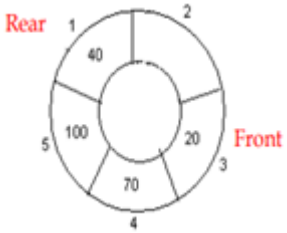


Fig 1.4.15 illustration of step10

11. Delete Front(20), Rear = 1, Front = 4.

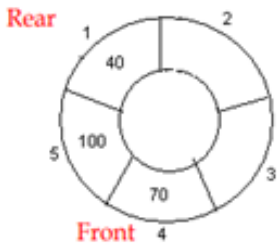


Fig 1.4.16 illustration of step11

12. Delete Front(70), Rear = 1, Front = 5.

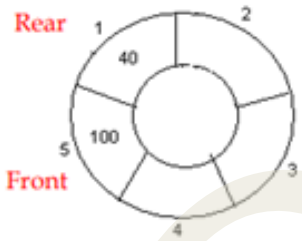


Fig 1.4.17 illustration of step12

♦ **Examples of Circular Queue:**

A traffic light sequence on a large roundabout is an example of a circular queue. It changes the signals circularly in a sequence with an equal interval of time. Some other real-time examples are the print spooler of the operating system, bottle-capping systems in cold drink factories, a routine of human life and days in a week.

1.4.2.3 Double-Ended Queue

Figure 1.4.18 shows the double-ended queue (Deques) graphical image. It is the third type of data structure in the queue. A double-ended queue is a linear list in which elements can be added or removed at either end but not in the middle. The double-ended queue is also called D-queue, DE-queue or deque, depending upon the operations. The double-ended queue operations are categorised into two types: input-restricted deque and output-limited deque.

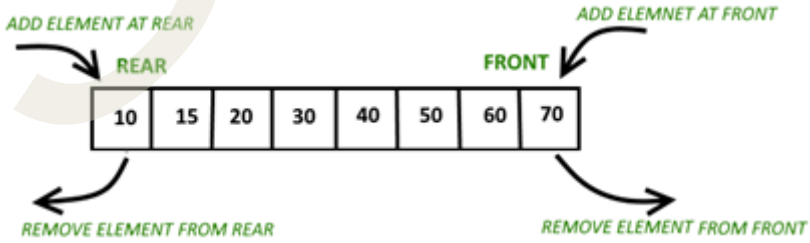


Figure 1.4.18 Double Ended Queue

1. Input Restricted Deque

As the name suggests, in an input-restricted queue, insertion operations are limited to

one end, while deletions can occur from both ends. Thus, data can be inserted at one end (say REAR end) only, but data can be deleted from both ends, as shown in Figure 1.4.19.

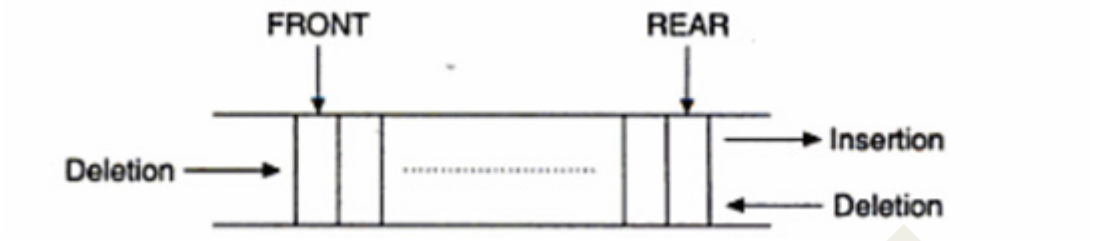


Fig 1.4.19 The Input Restricted Deque

2. Output Restricted Deque

An output-restricted Deque is a type of double-ended queue where deletions take place at one end only, say (FRONT end), but allows insertions at both ends. Figure 1.4.20 represents an output-restricted deque.

Fig 1.4.20 The Output Restricted Deque

♦ **Examples of the double-ended queue**

One of the real-world applications of the double-ended queue is storing a web browser's history. Recently visited URLs are added to the front of the deque, and the URL at the back of the deque is removed after some specified number of insertions at the front. Another common application of the deque is storing a software application's list of undo operations.

One of the real-world applications of the double-ended queue is storing a web browser's history. Recently visited URLs are added to the front of the deque, and the URL at the back of the deque is removed after some specified number of insertions at the front. Another common application of the deque is storing a software application's list of undo operations.

1.4.2.4 Priority queue

The priority queue is the fourth type of data structure. It is an extension of the queue data structure. This data structure is a particular type in which each element has been assigned a value, called the priority of the component, and an element can be inserted or deleted not only at the ends but at any position on the queue. Figure 1.4.21 shows a

priority queue.

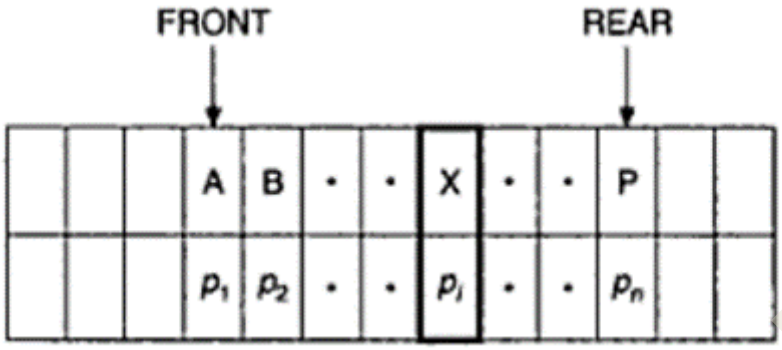


Fig 1.4.21 Priority Queue

With this structure, an element x of priority

p_i It may be deleted before an element that is at FRONT. Similarly, the insertion of an element is based on its priority; that is, instead of adding it after the REAR, it may be inserted at an intermediate position dictated by its priority value. A priority queue does not strictly follow the first-in-first-out (FIFO) principle, which is the basic principle of queue data structure. Instead, the priority queue operates and arranges the elements so that 'high-priority elements are served before the 'low-priority elements.

A priority queue can be implemented in many ways. These are:

1. Using Array
2. Multi-queue implementation
3. Using a linked list
4. Using heap tree

In this unit, we will discuss the array implementation of the priority queue.

1.4.2.5 Priority queue implementation using Array

In this representation, an array is used to hold elements and priority values of a priority queue. The elements are inserted at the REAR end. The deletion operation is performed in either of the following ways.

a. Traverse array of elements for the highest priority value, starting from the FRONT pointer. Then, delete the highest priority element from the queue. If the deleted element is not at the FRONT pointer, then shift all the remaining aspects after the deleted elements one position backwards to fill up the vacant position in the queue, as shown in Figure 1.4.22.

This method of deleting elements from the priority queue is inefficient because it requires continuous searching for the highest priority element from the queue.

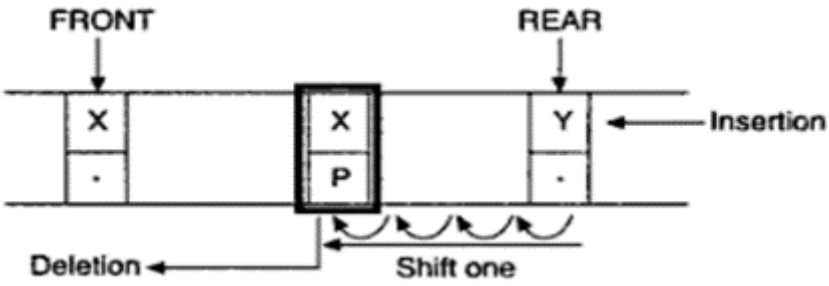


Fig 1.4.22 Delete operation in the priority queue

This method of deleting elements from the priority queue is inefficient because it requires continuous searching for the highest priority element from the queue.

The second method is to add elements at the REAR end and, using a stable sorting algorithm, sort the complete queue so that the highest priority element is at the FRONT end. When deletion of an element is required, then delete the element from the FRONT end only. As shown in Figure 1.4.23

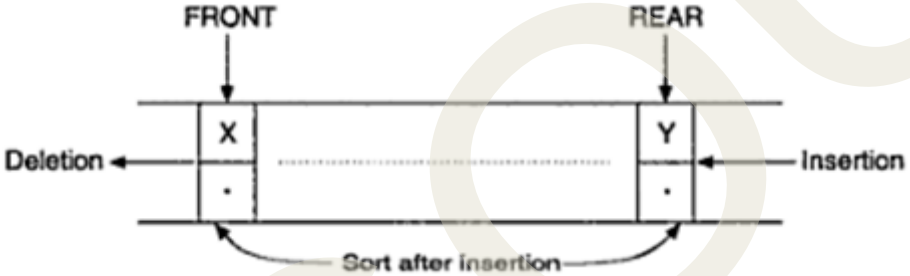


Fig 1.4.23 Another method of deleting an element in the priority queue

♦ **Examples of priority queue**

There are several real-world applications of priority queues. First, they are part of most operating systems. In an operating system, programs are chosen to run based on their priority. Many network routers also use a priority queue to determine which packet to send next. The high-priority traffic goes first in the priority queue.

1.4.3 Applications of Queue

The queue is an abstract data structure that is similar to stacks. Unlike stacks, a queue is open at both ends. One end is always used to insert data (enqueue), and the other end is used to remove data (dequeue). Applications of the queue are discussed below:

1. Requests can be served on a single shared resource, like a printer, CPU task scheduling, etc.

2. In real-life scenarios, call centre phone systems use queues to hold people calling them in any order until a service representative is free.

3. Handling of interrupts in real-world systems. The interrupts are handled in the same order as they arrive: first come, first served.

4. Operating systems often maintain a queue of ready-to-execute processes for a particular event to occur.

- Semaphores
- FCFS (first come, first serve) scheduling, for example, FIFO queue
- Spooling in printers
- Buffer for devices like keyboard
- The resource is shared among multiple consumers with the help of a queue. Examples include CPU scheduling and disk Scheduling.

Example:

The process of airport port simulation is the sharing of a single runway of an airport for both landing and take-off flights, which is an application of queue data structure. In print spooling, documents are loaded into a buffer, and then the printer pulls them off the buffer at its own rate. Spooling also lets you place several print jobs in a queue instead of waiting for each one to finish before specifying the next one.

1.4.4 Introduction to Recursion

The best way to solve a problem is by solving a smaller version of the same problem first. Recursion is a technique that solves a problem by solving a minor issue of the same type and turning it into smaller varieties of itself. The theory of recursion is established that a problem can be solved much more easily and in less time. Mathematicians often use recursive algorithms to solve many mathematical problems, such as factorial, powers, and greatest common divisor. In a programming language, recursion is the process of calling a function by itself. There are two types of recursions for solving problems: direct recursion and indirect recursion.

Direct recursion is the more straightforward way of solving problems; it involves a single step of calling the original function. The explicit recursion can be used to name just a single function by itself. The indirect recursion involves several steps. The indirect recursion (or mutual recursion) occurs when a function calls another function, resulting in the original function being called again.

1.4.4.1 Recursion Process in C language

The process of repeating the items similarly to how it was before is known as recursion. The programming language C is supporting the recursion process. Two conditions are critical for implementing recursion in the C language. The first is an exit condition, and the second is changing the counter. The exit condition helps the function identify when to exit that function. If the programmer does not specify the exit condition, then the code will enter into an infinite loop. Changing the counter condition works in every call to that function. The C language supports two types of recursive function calls: tail and non-tail recursion.

1. Tail Recursion

If a recursive function calls itself and that recursive call is the last statement in the function, it's known as tail recursion. Tail recursion is a form of linear recursion. In the tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. The tail recursive functions can often be easily implemented iteratively. A good example of a tail recursive function is given below:

Example: factorial of number 5

(factorial 5)

= (* 5 (factorial 4))

= (* 5 (* 4 (factorial 3)))

= (* 5 (* 4 (* 3 (factorial 2))))

= (* 5 (* 4 (* 3 (* 2 (factorial 1)))))

= (* 5 (* 4 (* 3 (* 2 (* 1)))))

= (* 5 (* 4 (* 3 (* 2))))

= (* 5 (* 4 (* 6)))

= (* 5 (* 24))

= 120

Answer: 120.

The tail recursive methods are easy to convert to iterative. The smart compilers can detect tail recursion and convert it to iterative optimised code. The tail recursion method is used to implement loops in languages that do not support loop structures explicitly.

2. Non-Tail Recursion

Operations are pending after the recursive call, and the recursive function call is done in the middle of the function. These types of functions are called non-tail recursion. An example of a non-tail recursion function is given below:

Input number: 4

Output: Factorial is: 24

Explanation: 1 * 2 * 3 *4 = 24

The factorial of number 4 can be implemented as a non-tail recursive function,

fact (4)

=(* 4 (factorial 3))

= (* 4 (* 3 (* 2 (factorial 1))))

= (* 4 (* 3 (* 2 (* 1))))

= (* 4 (* 3 (* 2)))

Answer = 24

1.4.4.2 Properties of Recursion

The recursive function can run infinite iterations like a loop. To avoid the endless running of a recursive function, the recursive function must obey two properties. They are based on criteria and a progressive approach.

- **Base criteria**

There must be at least one base criteria or condition, such that when this condition is met, the function stops calling itself recursively.

- **Progressive approach**

The recursive calls should progress so that each time a recursive call is made, it comes closer to the base criteria.

Finding the factorial of a number is an example of a recursive function. Evaluate 5! Table 1.4.1 shows all the values in a 5! Calculations.

Table 1.4.1: Calculate 5!

1	5	=	5	*	4															
	!				!															
2					4	=	4	*	3											
					!				!											
3									3	=	3	*	2							
									!				!							
4													2	=	2	*	1			
													!				!			
5																	1	=	1	*	0
																	!				!
6																					0	=	1
																					!		

In the 5! Calculation shown in table 1.4.1, each iteration of the factorial is reduced to a smaller one until the table reaches iteration0! which is defined directly as value = 1. In line 6, the value one is determined now and not as the factorial of another number. Finally, backtrack the entire process from line 6 to line 1.

The recursion function has the following advantages and disadvantages. The main advantage of a recursive approach is in the algorithm design. The recursive process allows programmers to take advantage of the repetitive structure present in many problems. The recursion can lead to more readable and efficient algorithm descriptions. Finally, recursion is also a valuable way to define objects with a similar repeated structural form.

The recursive function has the following limitations. It slows down the execution time and stores on the run-time stack more things than required in the recursive approach. If recursion is too deep, then there is a danger of running out of space on the stack. Some recursive functions repeat the computations for some parameters, and then the run time can be prohibitively long, even for straightforward cases.

1.4.5 Tower of Hanoi Problem

The tower of Hanoi is an example of a non-numeric recursion problem-solving process. It was invented by the French mathematician Edouard Lucas in 1883. The recursion technique can be applied to solve any of the wide variety of problems whose solutions are inherently recursive.

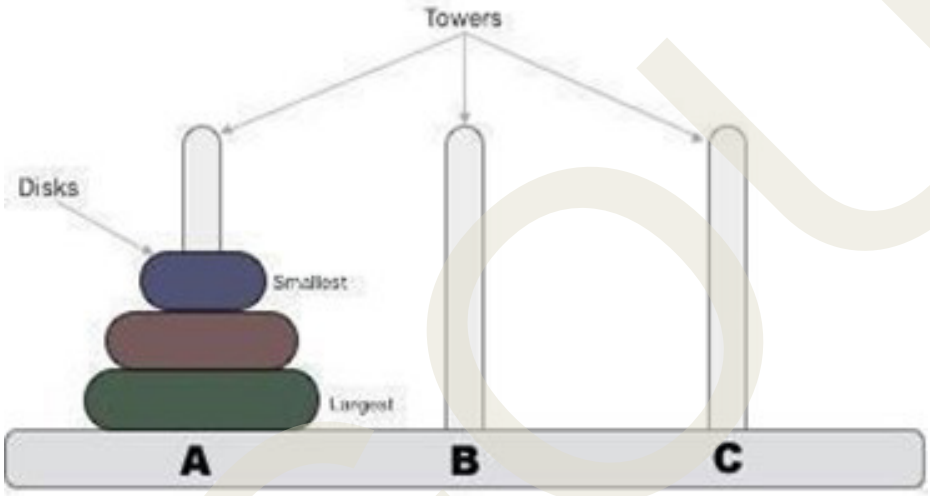


Fig 1.4.24 Tower of Hanoi

From Figure 1.4.24, the Tower of Hanoi problem is a mathematical puzzle which consists of three rods and n disks. The objective of the tower of the Hanoi Puzzle is to move the entire stack to another rod and, at the same time, obey the following simple rules. The rules are listed below:

1. Only one disk can be moved at a time.
2. Only the "top" disk can be removed. i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No large disk may be placed on top of a smaller disk.

Figure 1.4.25 contains three rods labelled 1,2 and 3. On rod 1, three different sizes and colours are in order: the giant ring on the bottom and the smallest one on top of the ring. The second and third rods, 2 and 3, are empty. The task is to move the three rings from rod 1 to rod 3 by successively moving a ring from one rod to another rod that is empty or has a larger diameter ring on top. To solve the tower of Hanoi puzzle that contains three disks, the stack of disks has to be shifted from rod 1 to rod 3 by abiding by the set of rules that have been mentioned above.

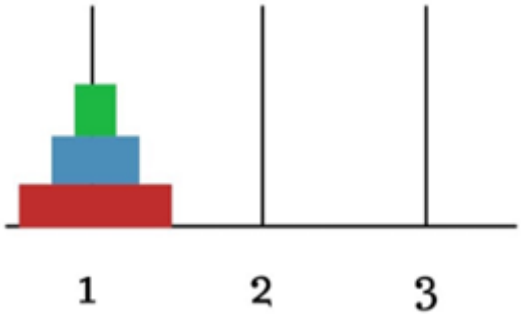


Fig 1.4.25 Illustration for Three Disks

Step 1: The smallest green disk, the uppermost disk on the stack, is shifted from rod 1 to rod 3.

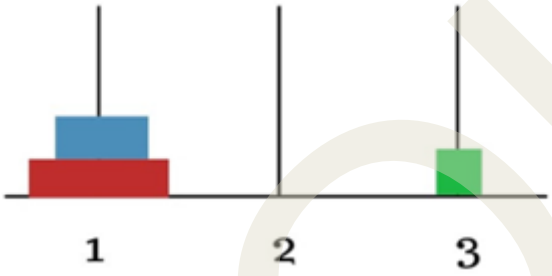


Fig 1.4.26 Illustration of Step 1

Step 2: Next, the uppermost disk on rod 1 is the blue-coloured disk shifted to rod 2.

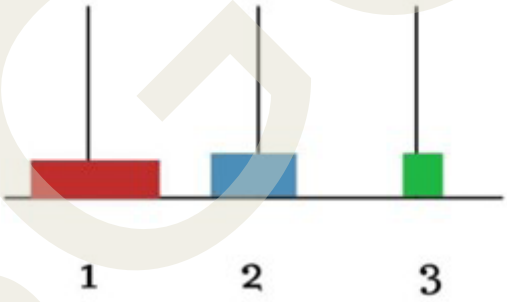


Fig 1.4.27 Illustration of Step 2

Step 3: The smallest disk placed on rod 3 is shifted back to the top of rod 2.

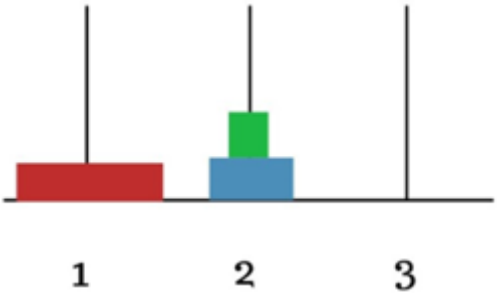


Fig 1.4.28 Illustration of Step 3

Step 4: Now, the largest red disk is allowed to be shifted from rod 1 to its destination, rod 3.

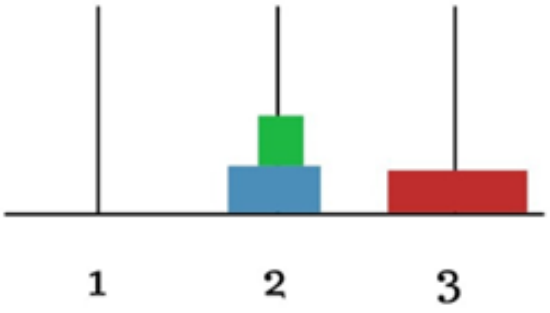


Fig 1.4.29 Illustration of Step 4

Step 5: Now, the two disks on rod 2 have to be shifted to their destination, rod 3 on top of the red disk, so first, the smallest green disk on top of the blue rod is moved to rod 1.

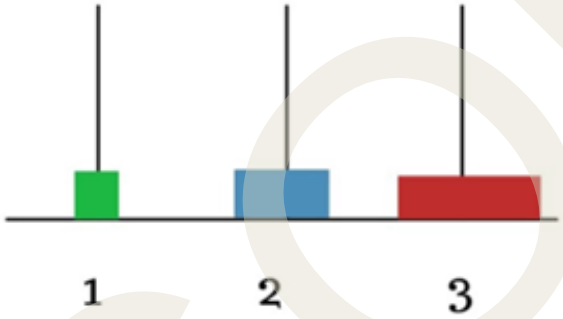


Fig 1.4.30 Illustration of Step 5

Step 6: Next, the blue disk is permitted to be shifted to its destination rod 3 stacked on to the top of the red disk.

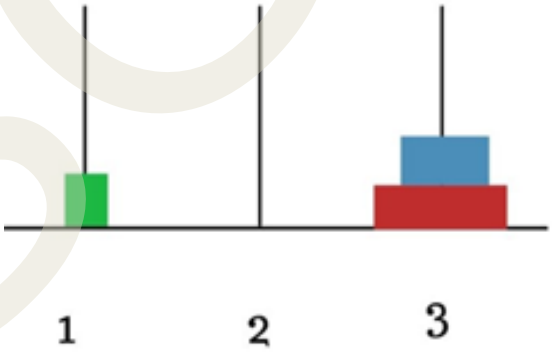


Fig 1.4.31 Illustration of Step 6

Step 7: Finally, the smallest green-coloured rod is also shifted to rod 3, which would now be the uppermost rod on the stack. So, the tower of Hanoi for three disks has been solved.

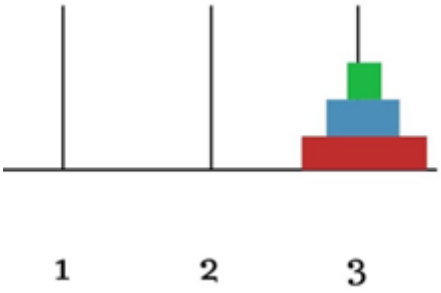


Fig 1.4.32 Illustration of Step 7

Recap

- The simple queue is a standard queue where insertion occurs at the FRONT of the queue, and deletion occurs at the END of the queue.
- In a Double Ended Queue, insertion and deletion operations can be done at both FRONT and END of the queue.
- Input restricted deque allows insertion at only one end (REAR end only) and provides deletion from both ends.
- Output restricted deque allows deletion from only one end (FRONT end only) and provides insertion from both ends.
- A priority queue is a particular type of queue in which each element is associated with a priority and is served according to its priority.
- The process in which a function calls itself directly or indirectly is called recursion, and the corresponding function is called a recursive function.
- Recursion is used to solve problems involving iterations in reverse order.
- Recursive solution is always logical, and it isn't easy to trace.
- A function calls itself. It's known as direct recursion.
- If f1 and f2 are two functions, the function f1 calls the other function f2, and the function f2 calls the function f1, which is indirect recursion.
- The objective tower of the Hanoi problem is to move the stack of disks from the initial rod to another rod based on the following rules:
    - A disk cannot be placed on top of a smaller disk.
- The queue data structure is used in different application areas such as resource sharing, data transfer, operating systems, and networks.

Objective Type Questions

1. What is another name for a circular queue?
2. What is the data structure in which elements can be inserted or deleted from both ends but not in the middle?
3. What does DEQUEUE refer to in a circular queue?
4. Which principle is based on a circular queue?
5. Which part is used to get the front (first) element from the circular queue?
6. Which data structure allows deletions at one end of the list but insertion anywhere?
7. The queues and the stacks can be implemented using either arrays or linked lists. Is this given statement true?
8. To input a list of values and output them in order, you could use a queue. Is the given statement true?
9. To input a list of values and output them in the opposite order, you could use a stack. Is the given statement true?
10. What is a linear list in which insertion and deletion are made from either end of the structure?
11. Identifies the data structure that allows deletions as per the priority.

Answers to Objective Type Questions

1. Ring Buffer
2. Deque(double ended queue)
3. Removing an element from the front of the queue.
4. FIFO (First In First Out)
5. FRONT
6. Output restricted queue
7. True

8. True

9. True

10. Deque

11. Priority queues

Assignments

1. What are the operations performed with the priority queue?
2. Write a few applications for the priority queue.
3. Mention the overflow and underflow conditions of the circular queue.
4. What is the benefit of the circular queue over the linear queue?
5. What is the application of a circular queue?
6. Comparing and finding out the common difference between a queue and a circular queue.
7. Write a C program to implement a queue using an array.
8. Explain the Tower of Hanoi Problem with suitable diagrams
9. Explain the array representation of the priority queue
10. Explain input-restricted and output-restricted queue

Suggested Reading

1. Sharma, A. K. "Data Structures using C", 2e. Pearson Education India, 2013.
2. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. "Data structures and algorithms." Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.
3. Weiss, Mark Allen. "Data structures and algorithm analysis." Benjamin-Cummings Publishing Co., Inc., 1995.
4. https://sonucgn.wordpress.com/wp-content/uploads/2018/01/data-structures-by-d-samantha.pdf

BLOCK 2

Linked List

Linked Allocations

Learning Outcomes

By the completion of this unit, the learner will be able to:

- introduce the concept of Linked List
- familiarise the representation and implementation of Linked List
- make them understand Self-referential structures.
- familiarise traversal of Linked list.

Prerequisites

How can you efficiently process data so that you will do lots of things with that data? How can we store the data in a way that makes it easier to retrieve and process? That is why we have the concept called Data structures.

Data structure is all about how you can structure your data so that you can store it and use it efficiently. You can store your data in such a way that it will be easier for your work in future. As an example, take the concept of a list; if we want to save data in sequential format, we actually use a list. A list is an abstract data type. Array, Stack, and Queue are examples of other abstract data types.

The figures below show some items in groups. Each group follows some grouping strategy.

Fig 2.1.1 Queue

Fig.2.1.1. **Queue** shows a new person will join at the end of the queue.

Fig 2.1.2 Stack of Books

Fig 2.1.2. In the figure, upon the Stack of books, a new one can be placed at the top.

Fig 2.1.3 Heap of stones

Fig.2.1.3. Heap of stones show that they are dumped without any specific order.

Fig.2.1.4 Egg Rack

Fig.2.1.4 shows an array of eggs in a rack. It shows that the egg can be placed at any position in the tray. The concept of data structure is similar to the collection of items in the above pictures.

In this unit, we are going to talk about a new data structure called Linked List.

Consider a movie theatre where the seats are arranged like arrays. The seats are allocated only in the order of seat numbers, allocated only sequentially. Suppose, after all the tickets were sold, some people cancelled the tickets, and some people came to buy the tickets. However, tickets cannot be issued because the tickets are allocated only in the order of seat numbers. This is an unfair situation. Seats cannot be assigned even if they are free. This actually happens in the allocation of arrays. Even if ten vacant positions are available in the memory, the array can only be allocated if the vacant spaces are contiguous. There comes the role of a Linked list.

In a Linked list, data are connected by links rather than by contiguous memory locations.

Key Concepts

Data Structure, Linked list, Singly Linked list, Circular linked list

Discussion

2.1.1 Linked List

Now, we have to discuss in detail the types of linked lists and how data is stored in a linked list.

Suppose we want to store the names of students in a memory. There are two available ways for us to maintain that list in the memory. Namely:

- An Array
- A Linked List

You have already learned about arrays. An Array is a static data structure where elements are stored in contiguous memory locations, and there is a limit on the number of items to be stored.

However, a Linked list is a dynamic data structure where there is no limit to the number of items. Unlike arrays, elements in the Linked list are scattered in the memory, but each data is linked with pointers (pointer points to the address of the memory location where the data resides). The linked list grows when a new data item is added and shrinks when it is removed from the list. We will see the different types of Linked lists in this unit.

- **Singly Linked List, Doubly Linked List, Circular linked list** are some types of linked lists.

What is a singly Linked List?

Singly linked list is one of the basic types of linked lists. Here, navigation is forward only. That means we can go in a forward direction and can't go backwards. In doubly Linked list, forward and backward navigation is possible.

A singly linked list or a Linked list is made up of nodes that consist of two parts

- Data
- Link

What is a Node? A node looks like the following, as referred to in Figure 2.1.5

The data part contains the actual data, and the Link part contains the address of the next node in the linked list. So, a Linked list is a collection of nodes, where each node consists of data and a link, i.e., a pointer to the next node in the list. The first node of a linked list is called Head.

2.1.2 Representation of Linked List

Suppose we want to store a list of names of students in memory, such as Athul, Hima, Manu, and Rihan. So, we need to create four different Nodes.

Fig 2.1.6 Four node examples of student names

The first node stores the name 'Hima' in the data part of the node. The second node contains the data 'Manu' and so on. These nodes need not be stored in a sequential order. For simplicity, the address of each node is denoted by 100, 102, 104, 106. Since it is a linked list, the address is also not sequential. The first node address is 100, and the last node is 106.

The list is scattered in the memory. There is some way to reach the next node of the list to complete the list. In order to do that, we have to store the address of each node in the Link part of the node. The first node contains the address of the second node. The second includes the address of the third, and so on.

The link part of the last node contains NULL, a keyword which represents the end of the node. So, from the first node, we can reach the second node because the link part of the first node contains the address of the second node. In that way, we can traverse the whole list until we see a NULL in the Link part.

NULL in the link part of a node denotes that it is the end of the list.

But how do we get the address of the first node? For that, a pointer called START is used to point to the first node in the linked list. The address of the first node is contained in the START.

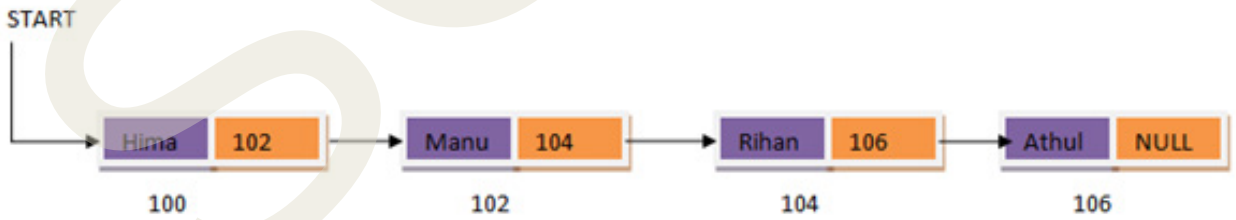


Fig.2.1.7 Four node example of Linked List

2.1.3 Implementation of LinkedList

How do you create a node for a linked list in C? A Linked list is a collection of nodes, where each node contains two portions: one is actual data, and the other is a pointer to the next node.

So, to implement a linked list in C, we need a Structure. We have already learned that structure is a user-defined datatype. The member of a structure can be a pointer, and it may be a pointer to the same structure. We call such a structure a self-referential structure.

How a Node is declared in C?

A self-referential structure is used to create a node for a Linked list.

A self-referential structure is a structure which contains pointers pointing to a structure of the same type.

For example,

```
struct sample {

int a; char b;

struct sample *self

}
```

Here, 'sample' is a structure that has a pointer **self** that points to the structure sample itself. So, it is referencing itself. Hence, it is called a self-referencing structure.

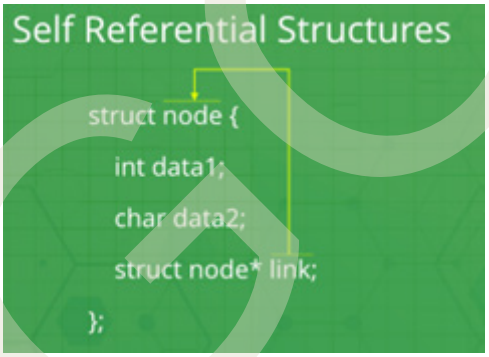


Fig 2.1.8 Self-Referential Structures

We have already seen that a node is a combination of two different types of data. It consists of data as well as link parts. To combine two different types into a single type, we use structure. Nodes in the linked list can be designed using the structure as follows.

```
struct node {

int data1;

char data2;

struct node *link;

  }
```

The structure struct node consists of three different types of data. One is integer data, and the second is character data. Data could be anything, and any number of data is

also possible. The data type could be anything like char, float or any other data type. The third element, link, is a pointer to the same structure type node. **What is the struct node *link?**

Link is a **pointer to some other node,** and as we know, a node is nothing but structure only. So, the link must be a pointer to the struct node. Hence, it is called a self-referential structure, as shown in Figure 2.1.8.

2.1.4 Algorithm for Creation of a complete Linked list

The creation of a linked list can be viewed as a repeated insertion operation at the tail of the linked list.

The following are the steps involved in the process.

Step 1: Create a node and collect the address of that node.

Step 2: Store data in the Data part of the node and NULL in the link part of the node

Step 3: Store the address of that node in the START if that node is the first node.

Step 4: If it is not the first node, store the address of the new node in the link part of the previous node.

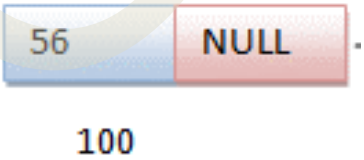
Step 5: Repeat the steps 1 to 4 till the user request stops.

2.1.5 Steps to create a Linked list

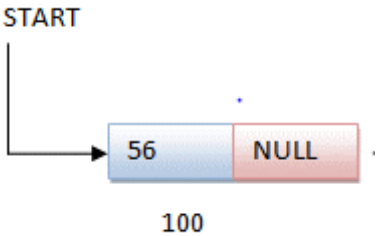
1. Linked list is empty

START

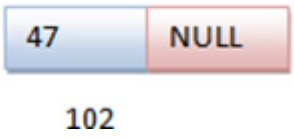
2. The first node is created with address 100. Data and links are filled



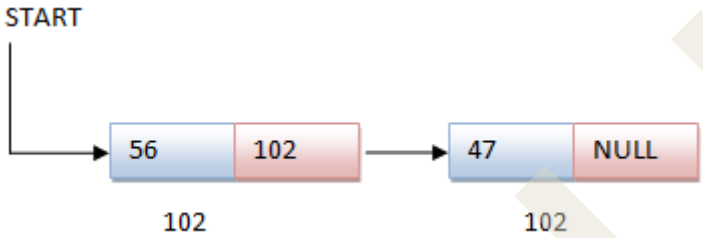
3. The address of the first node, here=100, is stored in the START



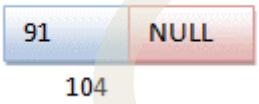
4. The second node is created with address 102. Data and link are filled



5. The address of the Second node is stored in the link part of the first node.



6. The third node is created with address 104, and Data and Link are filled.



7. The address of the **third node** is stored in the **link part of the second node.**

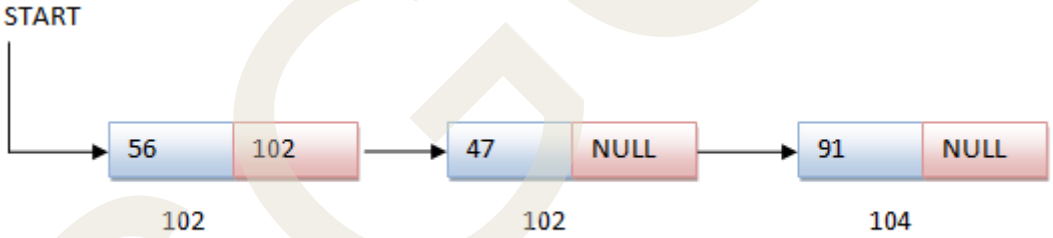


Fig 2.1.9 Creation of a Linked list

Stacks and queues can be implemented using Linked list, too, which results in dynamic stacks and queues. Here, we discuss the creation of a singly linked list. In the next Unit, we will discuss the insertion and deletion of nodes in the singly linked list.

Creating a node in C

```
#include <stdio.h>

        #include<stdlib.h> struct node{

        int data;
```

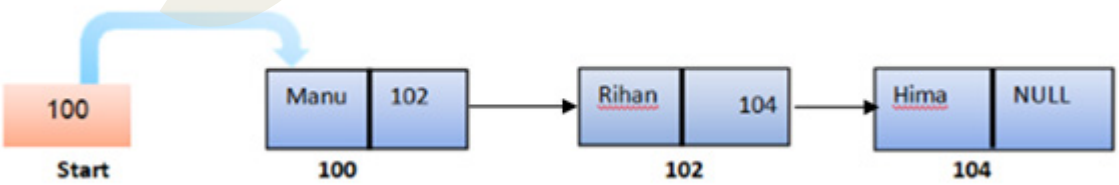
```
        struct node *link;

        };
int main( ) {

struct node*start=NULL;      //start is a special pointer that points to the first node. Here, initialise to NULL.

start = (node *) malloc(size of( node)); //malloc is a function for memory allocation for creating a node. The address of the first node is stored in the start pointer.

start →data=56;          // since it is a pointer, the data is accessed using an arrow pointer. Initialize data by 56

start→link =NULL;  //link part is filled with NULL

printf("%d",start→data);       //print the data on first node using the access pointer

return 0;

}
```

2.1.6 Traversing a linked list

Traversing a singly linked list means visiting each node of a linked list until the end node is reached. In the case of a linked list, traversal can begin from the first node. The 'Start' pointer gives the address of the first node so that we can access the data part using the arrow operator. From the link part of the first node, we get the address of the second node. We can access the data and link part of the next node with this address. This method is continued until a NULL pointer is found in the link of a node.

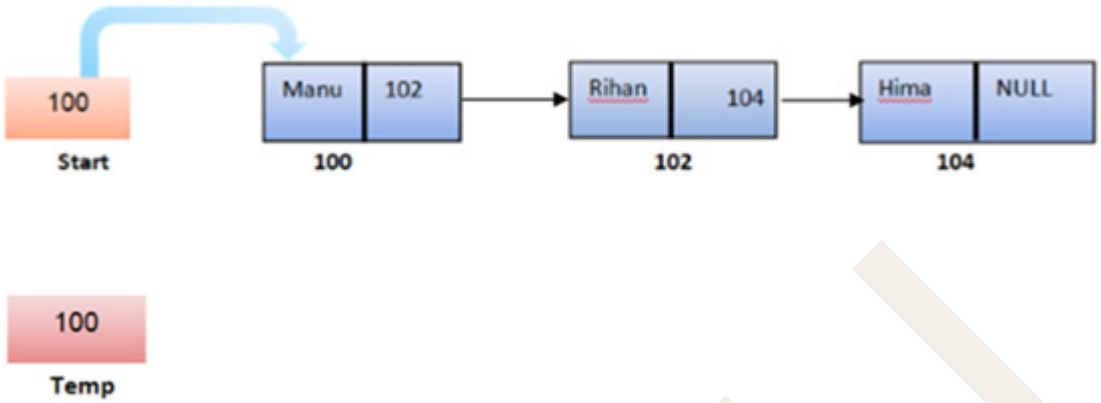
For the sake of simplicity, we will consider the following example. We assume that we already have a linked list, and we need to traverse this list. Not only do we need to traverse this list, but we also have to calculate the total number of nodes in this list by traversing this linked list. The figure below derives the steps required for traversal operation in a linked list. It is assumed that Temp is a Temporary pointer of the 'node' type and Val is a variable to store the data read from a node.

(1) Linked List having three nodes.

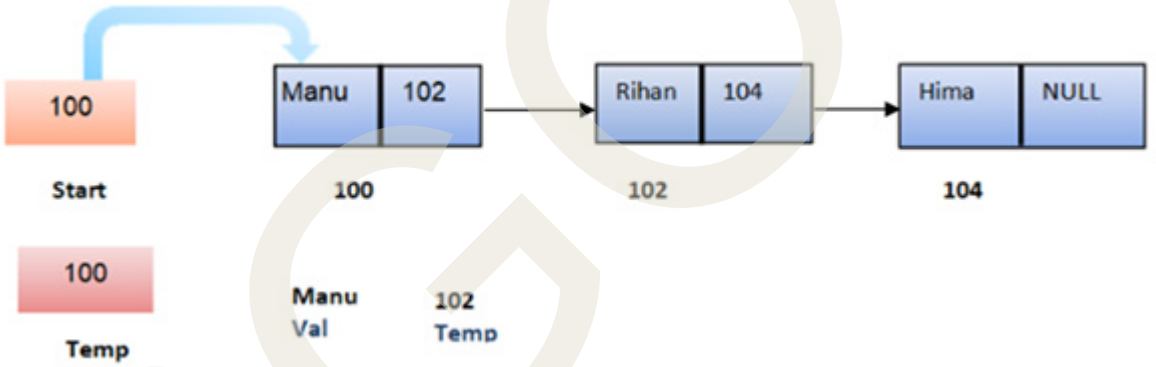


(2)The content of the start is copied into Temp(Because if we change the value of the start, the address of the first node will be lost when traversing). Now, Temp can point

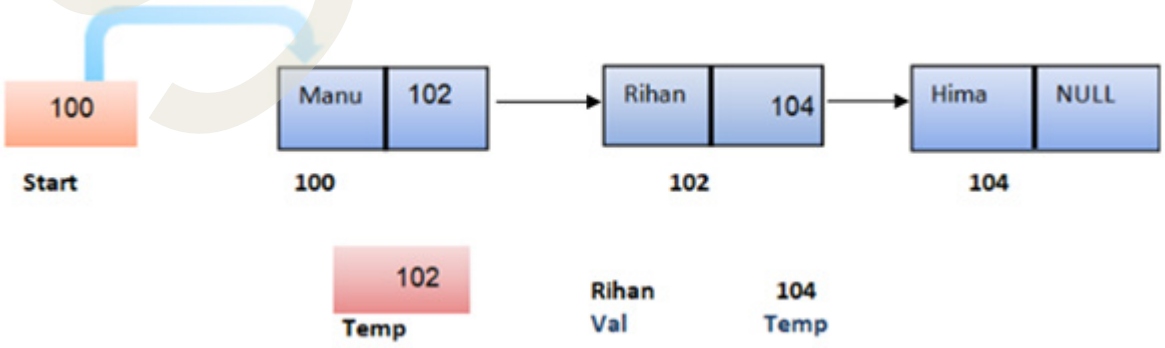
to the first node.



(3) The first node is accessed using Temp, and Data is retrieved to Val. After that, the Temp is updated by the address of the second node.



(4) The second node is accessed using Temp, and data is retrieved. After that, the Temp is updated by the address of the third node.



(5) Third node is accessed using value in Temp, and data is retrieved to Val. After that, the link part of the third node updates that Temp. Since it is NULL, traversal ends here.

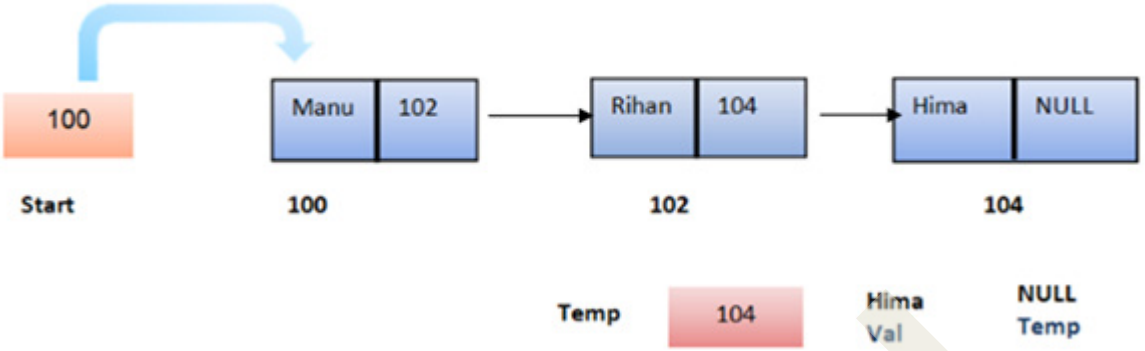


Fig 2.1.10 Showing traversal operation in a linked list

2.1.7 Algorithm for Traversal operation in linked list

Step 1: The address of the first node is taken from Start and stored in Temp.

Step 2: Using the stored address in Temp, find the data of the first or next node and store it in Val.

Step 3: The value of the link part of this node is found (or the address of the next node) and stored in Temp

Step 4: Repeat step 2 until the content of Temp is not NULL; else stop.

The steps explained above are needed in the creation of a linked list if Start is not NULL. In that situation, the last node's address is necessary to store the address of the new node in its link. The traversal operation begins from the first node, and the address is stored in a temporary pointer variable (Temp in the figure). It will update by copying the content of the link to the node pointed by Temp. Traversal will continue until the link of a node pointed by Temp shows a NULL value.

Recap

- A linked list is a linear collection of data elements where elements are not stored in contiguous memory locations.
- A linked list is a collection of nodes, where each node contains two portions. One is actual data, and the other is a pointer to the next node.
- Traversing a singly linked list means visiting each node of a linked list until the end node is reached.
- Self-referential structure is a structure which contains **pointers to a structure of the same type**.
- Structure contains a pointer, and by using that pointer, we store the address of the same structure.
- The address of the first node is stored in a pointer called Start.
- If there is no next node, a NULL value is stored in the link field of that node.

Objective Type Questions

1. What is the term that refers to the Address contained in the node?
2. Give an example of a dynamic data structure.
3. Name the structure which contains a pointer to itself.
4. What does the last node of the Linked list contain apart from the last data item?
5. In a linked list, what is each element commonly referred to as?
6. What is the process of visiting each element in a data structure called?
7. What is the primary advantage of a linked list over an array?
8. In linked list terminology, what is the first node of a list called?
9. During traversal, what pointer is used to move through the list?
10. A linked list contains a data part and ---------- part.

Answers to Objective Type Questions

1. Link
2. Linked list
3. Self-referential Structure
4. NULL
5. Node
6. Traversal
7. Better memory utilisation
8. Head
9. Temp
10. Link

Assignments

1. Explain the advantage of the Linked list over arrays.
2. What is a Self-referential structure? What is the use of a Linked list?
3. Explain how a linked list is implemented using C language.
4. Describe the traversal process of a singly list with the help of a neat diagram.

Suggested Reading

1. "Data Structure using C" by A K Sharma.
2. "Data Structures and Program Design in C" by Kruse Robert L.
3. "Data Structures and Algorithms Analysis in C" by Mark Allen Weiss.
4. "Data Structures and Algorithms" by Alfred V Aho and Jeffrey D Ullman

Operations on Linked List, Search and Sort, Linked List vs Array

Learning Outcomes

After completion of this unit, the learner will be able to:

- To understand the basics of node insertion in data structures.
- To familiarise various types of node insertion in the Linked List
- To make them understand how to delete a node in a Linked List.
- To discuss searching a node in a Linked list
- To compare the advantages and disadvantages of Linked List and Arrays.

Prerequisites

If the memory is allocated before the execution of a program, it is fixed and cannot be changed. We have to follow an alternative scheme to allocate memory only when it is needed. The data structure, called a Linked list, provides a more convenient and efficient storage system that does not require the use of arrays.

We can say a List is an ordered collection of data. An array is considered a list of data that can be accessed randomly by the use of an index.

A linked list is a collection of objects linked together by a reference. Object with data and reference is termed a node. The last node contains a NULL reference to show the end of the list. Unlike arrays where memory is allocated as a contiguous block of memory, in the linked list, the memory required for each node can be allocated as and when needed, thereby providing a better way to manage free memory.

Now, the question is, how can you insert a node at the beginning of the linked list? How can you insert a node at a position you wish to? How to search for an element in a linked list? Answers to these questions are going to be discussed in this unit.

Key Words

Linear Linked list, Node Insertion, Node Deletion, Linked list searching, Sorting

Discussion

2.2.1 Insertion in a linked list

Insertion of an item in a linked list is the process of placing the node containing the item in a particular position. To insert the nodes into a linked list, the following three things should be done

1. Allocating a node.
2. Assigning the data.
3. Adjusting the pointers.

As in the case of arrays, nodes can be inserted anywhere in a linked list –at the beginning, at the end or in between any nodes. i.e., inserting a new node into the linked list has the following three instances

1. Insertion at the beginning of the Linked list.
2. Insertion at the end of the Linked list.
3. Insertion at the specified position within the list.

2.2.1.1 Inserting a node at the beginning

In order to insert a new node at the beginning of the linked list, the following steps are to be followed.

1. If the linked list is empty or the node to be inserted appears before the starting node, then insert that node at the beginning of the linked list.

2. When a **new node** is created, the node can be inserted **at the beginning of the list** by copying the content of **Start** into the **link part of the new node** and the **address** of the new node **into Start**.

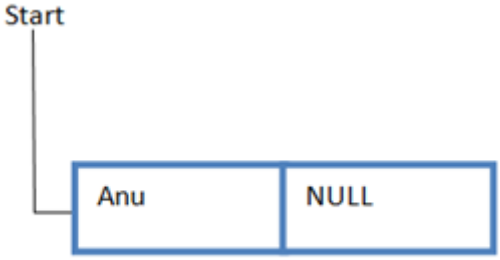
An algorithm for inserting the new node at the beginning of the linked list.

Step 1: Create a new node with a given value in the data field of the new node

Step 2: Check whether the list is Empty (Start==NULL),

Step 3: A new node is created. The address is stored in the Start pointer, and the Link of the new node is assigned as NULL

link [new_node] = =NULL



Step4: If the list is not empty,

then set link[new node]= Start.

Start = new_node

This means storing the address of the Start pointer (here 102) to the Link part of the new node and replacing Start with the address of the new node (i.e. 100, the address of the new node).

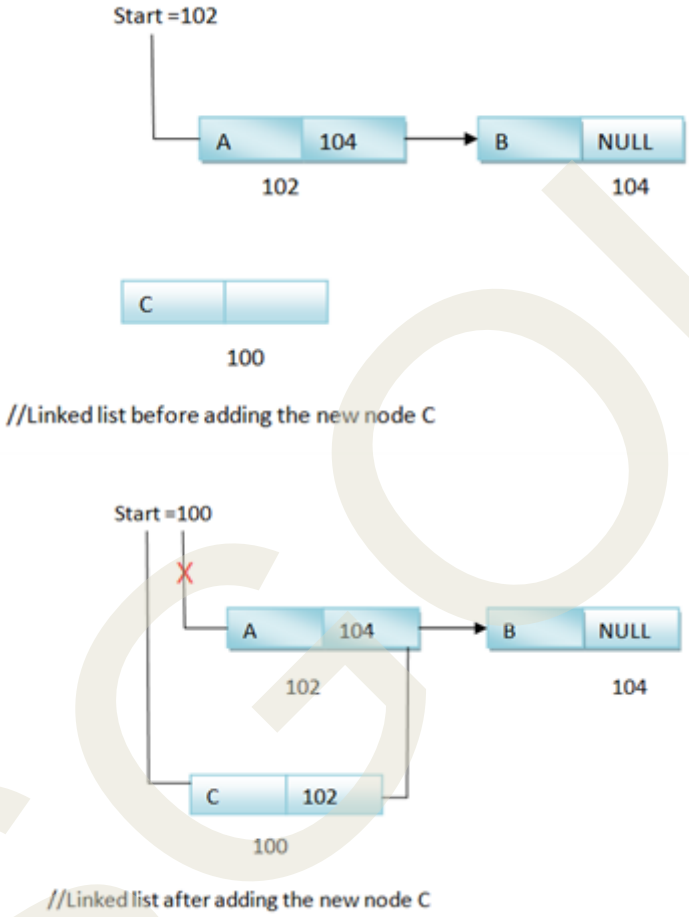


Fig 2.2.1 Inserting a node at the beginning of the linked list

2.2.1.2 Inserting at the End of the List

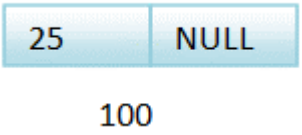
Similarly, if the node to be inserted appears after the last node in the linked list, then insert that node at the end of the linked list. To insert a node at the end of the list, we have to copy the **address** of the new node to the link part of the last node. Assign NULL in the link part of the newly added node.

We can use the following steps to insert a new node at the end of the single linked list.

Step 1: Create a node new_node with value in the data field (here, 25), and the link of new_node is NULL.

data[new_node]=25.

link[new_node]=NULL.



// A node termed new node is created with address100

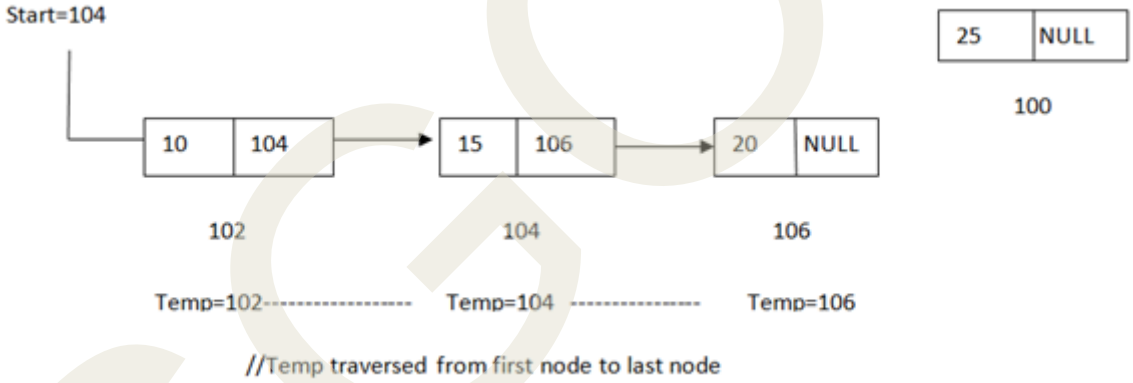
Step 2: Check whether the list is Empty (i.e., Start==NULL)

Step 3: If the list is empty, then set Start=new node

Start=100 **//Assigning the address of the new node to the start.**

Step 4: If it is not Empty, then a node pointer temp may be defined, and it will be initialised with Start.

Step 5: Keep moving the temp to its following node till it reaches the end node in the list (finding the link of the End node where link part=NULL)



Step 6: Make the link field of the last node point to the new node 100 and make the link field of the new node =NULL.

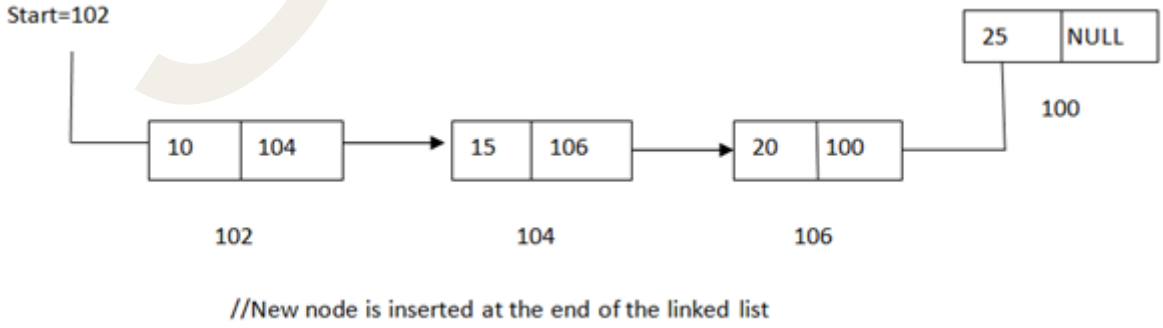


Fig 2.2.2 Inserting a node at the end of the linked list

2.2.1.3 Inserting a Specific Location in the List (in between two nodes)

The following steps are used to insert a new node in a specific location in the linked list. Here, we are trying to insert a node in the third position on the linked list. A series of operations are to be performed to insert the third position in a linked list. Let **temp**, **pre-node** and **post-node** be pointers of the Node type structure. Assume **POS** is a variable which contains the value of the position where the node is to be inserted. The following steps can be developed for the insertion operation.

Step 1: Create a node, and the address is stored in temp.

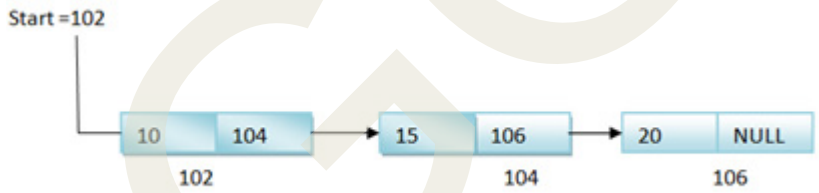
Step 2: Store the data and link part of this node using temp.

Step 3: By traversing the list, identify the location where the new node is to be inserted. Also, obtain the address of the nodes at position POS-1 and POS in the pointers **pre-node** and **post-node,** respectively (POS1 and POS are the positions of **pre-node** and **post-node)**.

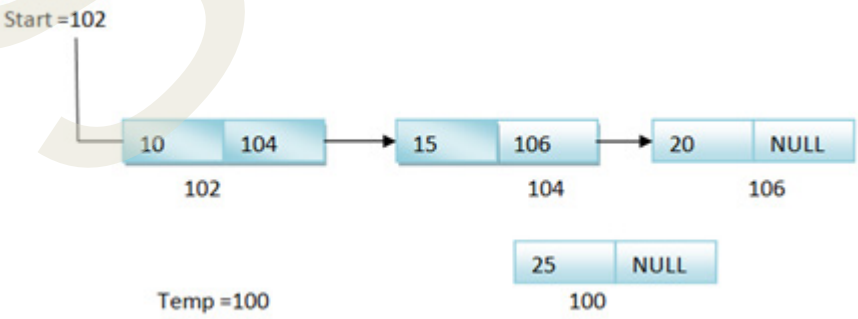
Step 4: Copy the content of the temp (address of the new node) into the link part of the node at position (POS-1), which can be accessed using prenode

Step 5: The content of the postnode is copied (address of the node at position POS) to the link part of the new node, which is pointed to by temp.

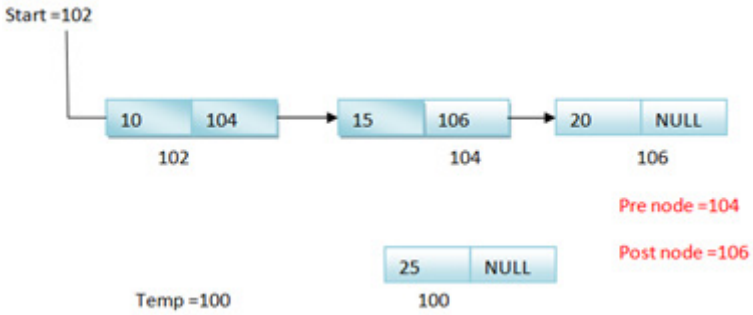
1. Linked list having three nodes. A new node is going to be inserted at the 3rd position.



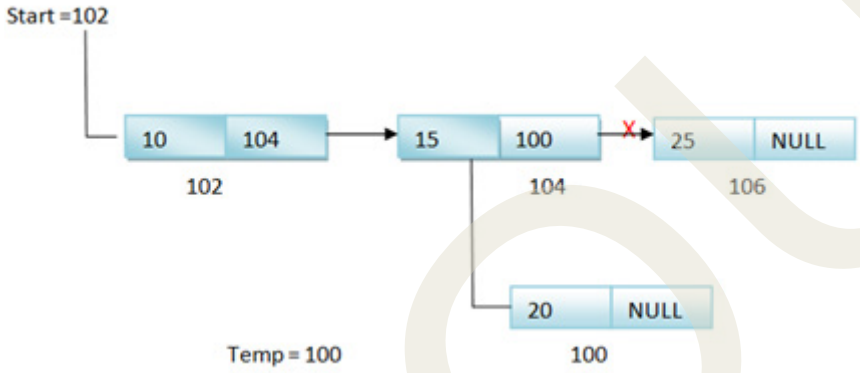
2. A new node is created, and the address is stored in temp.



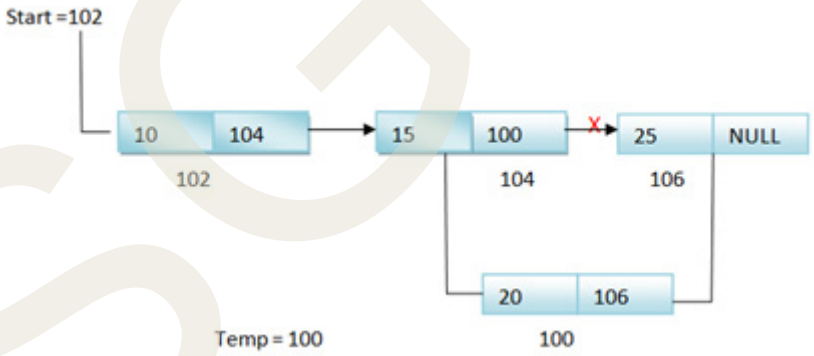
3. The address of the second node is copied into the prenode, and the address of the third node is copied into the postnode.



4. Address of the new node (from temp) is copied into the link part of the second node pointed to by the prenode, and the new node is linked.



5. The address of the fourth node available in the postnode is copied into the link list part of the new node.



6. Linked list after the insertion of a new node at the third position.

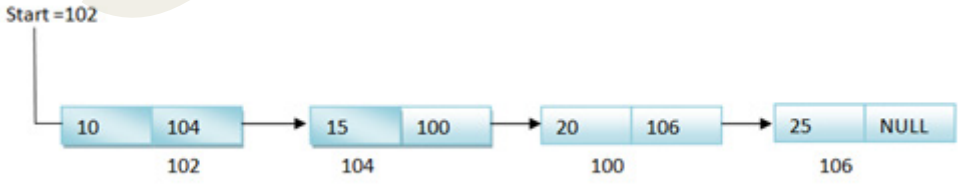


Fig 2.2.3 Inserting a node from a specific location of the linked list

2.2.2 Deletion from a linked list

Deleting an item from a linked list is the process of removing a node from the linked list. The position of the node to be removed will be given. Instead of this, if the data item is given, the node containing that item is to be searched, and its position is to be noted so that we can apply the steps for the removal operation. Any node, whether first, last or node at a specified position, can be removed from the list. To delete the first node, we have to **copy the content in the link part of the first node into Start**. The last node can be deleted by **assigning the NULL value to the link part of the second lastnode**.

Certain steps are to be performed to remove a node from a specified position, which will be explained further in this unit.

2.2.2.1 Deleting from the beginning of the list

We can use the following steps to delete a node from the beginning of the single linked list.

Step 1: Check whether the list is empty (Start ==NULL)

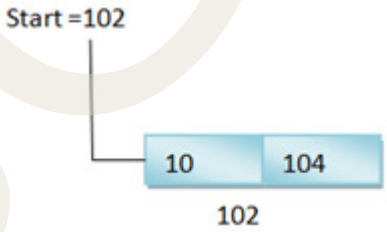
Step 2: If it is empty, then display 'List is Empty!!! Can't delete the node' and terminate the function.

Step 3: If it is not Empty, then Check whether the list has only one node

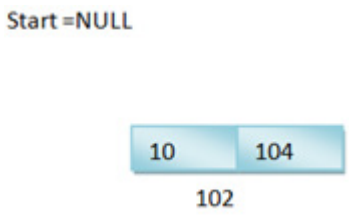
Step 4: If it is true, then set Start=NULL (Setting Empty list conditions) and stop.

Step 5: If the list contains more than one node, then copy the content in the link part of the first node into Start.

The list contains only one node

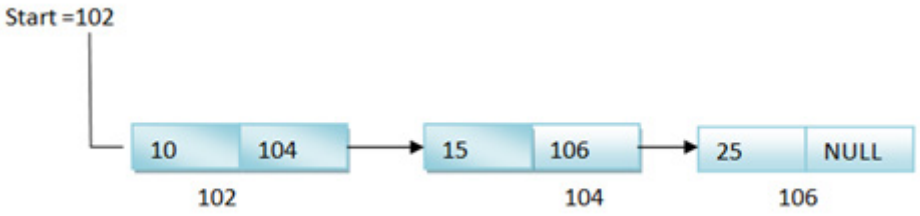


Just set Start = NULL



The node is removed from the list.

2. The list contains more than one node



Copy the content in the link part of the first node into Start.

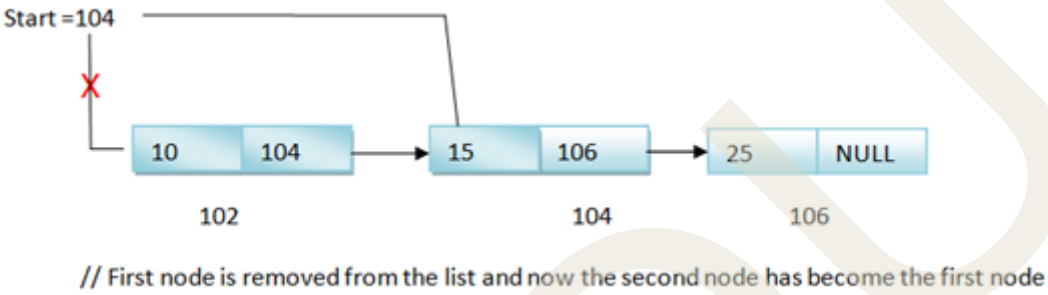


Fig 2.2.4 Deleting a node from the beginning of the linked list

The value of Start is now 104

2.2.2.2 Deleting from the End of the List

We can use these steps to delete a node from the end of the singly linked list.

Step 1: Check whether the list is empty (Start==NULL).

Step 2: If it is empty, then display 'Empty list..!!!

Cannot Delete and terminate the function

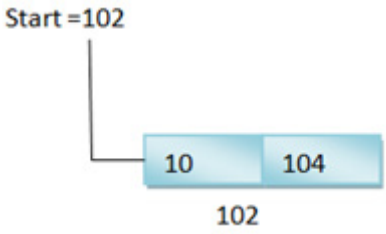
Step 3: If it is not Empty, then Check whether the list is having only one node

Step 4: If it is true, then set Start=NULL (Setting Empty list conditions) and stop.

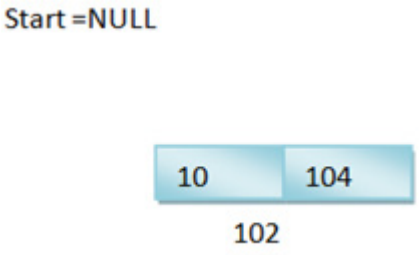
Step 5: If the list has more than one node, then traverse until the last node is reached. (Keep the address of the previous node in pre node while traversing).

Step 5: Set NULL value in the link field of the pre node. Then, the last node gets deleted from the list

1. List contains only one node.

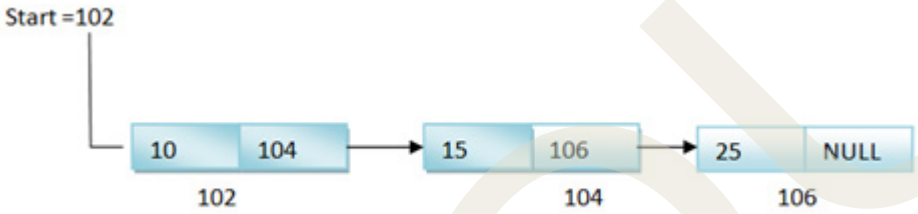


Just set Start = NULL

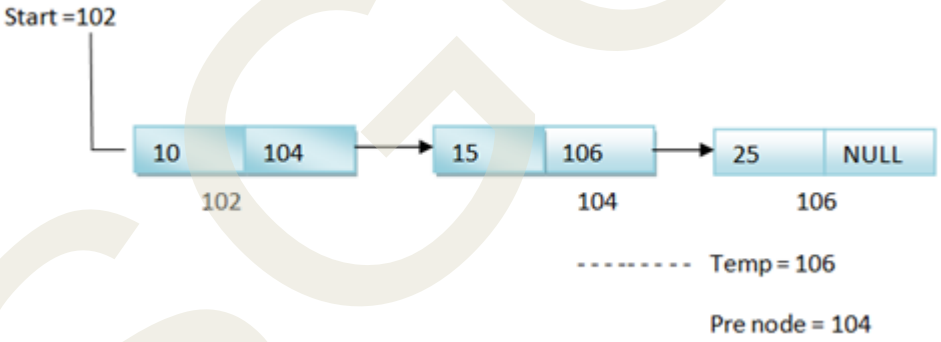


The node is removed from the list.

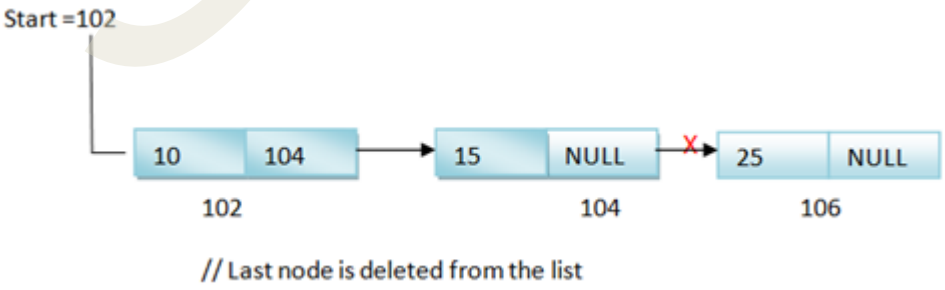
2. List contains more than one node



Traverse until the last node is reached. (Keep the address of the previous node in pre node while traversing).



Set NULL value in the link field of the pre-node



Linked list after the last node is deleted.

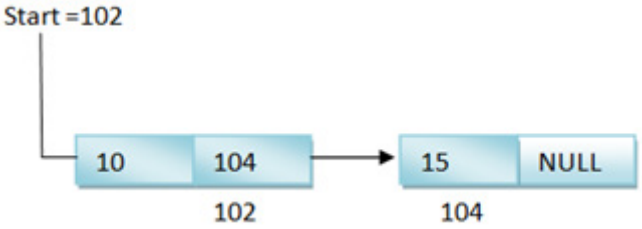


Fig.2.2.5 Deleting a node from the end of the linked list

2.2.2.3 Delete a Node in a Specific Location from the List

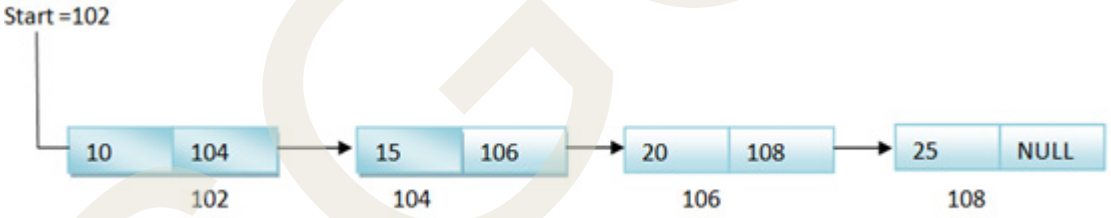
To remove a node from a specified position, all these steps are to be performed. The figure below describes the procedure for the removal of the third node from the linked list, which initially had four nodes. It is assumed that prenode and postnode are pointers of Node type structure. Consider **POS** as a variable where the variable contains the position of the node to be removed. Following steps are involved in the deletion operation

Step 1: Obtain the address of the nodes at the position, POS-1 and POS+1, in the pointers **pre–node** and **post-node,** respectively, with the help of a traversal operation.

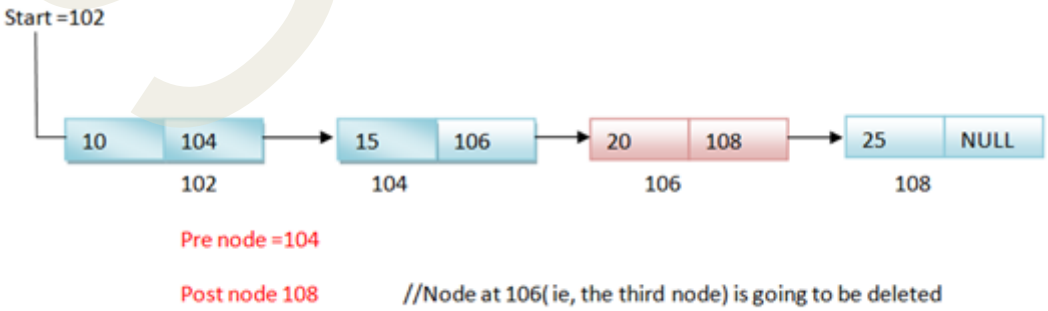
Step 2: Copy the content of the postnode (address of the node at position POS+1) into the link part of the node at position (POS-1), which can be accessed using prenode.

Step 3: Free the node at position POS.

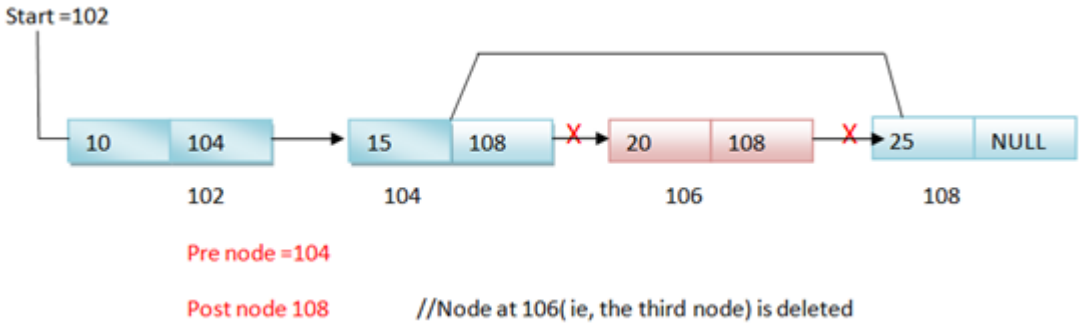
a. Linked list having four nodes



b. The address of the second node is stored in the prenode, and the address of the fourth node is in the postnode



c. Address of the fourth node available in the post-node is copied into the link part of the second node pointed to by the prenode. Thus, the third node is removed from the list.



d. Linked list after the deletion of the third node.

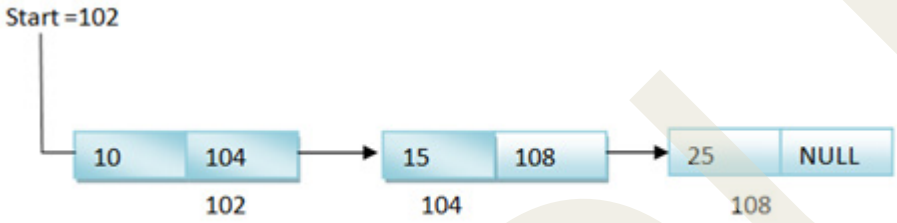


Fig 2.2.6 Deleting a node from a specific location of the linked list.

If we apply the first two steps in the linked list, even after deleting the third node from the second node, the presence of the third node will still be there in the memory, pointing to the fourth node. So, it should be freed using the memory deallocation facility provided by the programming language.

While implementing operations in the linked list, temporary pointers like temp, prenode, postnode, etc, should also be freed after the operations.

2.2.3 Searching in a Linked list

Searching in a linked list involves traversing through the nodes of the list to find a specific value.

Step-by-Step Process of Searching in a Singly Linked List

1. Access the Head Node:// Start is also referred to as Head

- Start the search at the head of the linked list.

2. Traverse the List:

- Initialize a pointer (often called Temp) to the head node.
- Check the value of the Temp node:
  - If it matches the search value, the search is successful.
  - If it does not match, move the Temp pointer to the next node.
- Repeat this process until you find the value or reach the end of the list.

3. Check for End of List:

- If the Temp pointer becomes NULL, it means the end of the list is reached, and the search value is not found.

2.2.3.1 Example

Consider a linked list with the following nodes:

Head → 10 → 20 → 30 → 42 → 50 → NULL

You want to search for the value 42.

Step-by-Step Execution

1. Initialize the Pointer:

Temp = Head (which contains 10).

2. Traverse and Compare:

Check Temp. data (ie,10). It does not match 42.

Move Temp to the next node.

Check Temp. data (ie,20). It does not match 42.

Move Temp to the next node.

Check Temp. data (ie,30). It does not match 42.

Move Temp to the next node.

Check Temp. data (ie,42). It matches the search value.

3. Result:

Value 42 is found in the linked list.

2.2.4 Sorting in Linked List

Sorting a linked list means arranging the elements in an order(either in ascending or descending order in the case of integer elements). Any sorting algorithm can be used to sort a linked list. Here, we are discussing the Selection sort.

The main idea of selection sort is to repeatedly find the minimum element from the unsorted part and move it to the sorted part.

Steps:

1. Initialize the sorted and unsorted parts of the list.

2. Find the minimum element in the unsorted part.

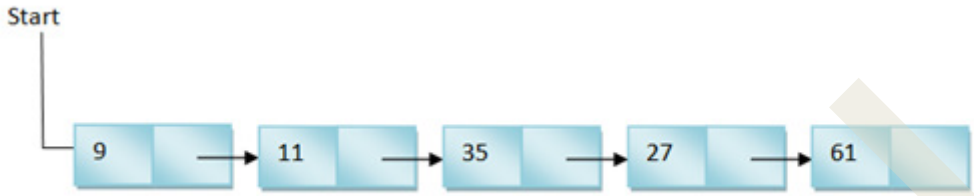
3. Move the minimum element to the end of the sorted part. That is, Swap the minimum element with the element at the end of the list for the first iteration, with the second last

element in the second iteration and so on.

4. Repeat until the entire list is sorted.

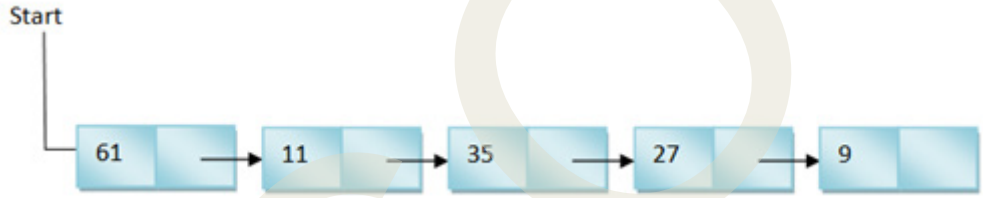
2.2.4.1 Example

Consider the list = 9 11 35 27 61



We have to sort this list in descending order. For that, perform the following steps:

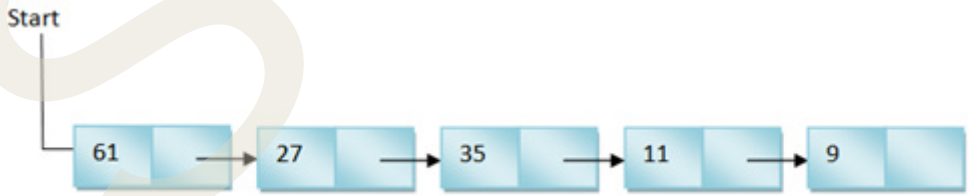
Find the minimum element in the list (from 9 to 61) and swap it with the element at the last position.



// 9 and 61 are swapped

Now, the element at the last position (9) is sorted and placed in its right position.

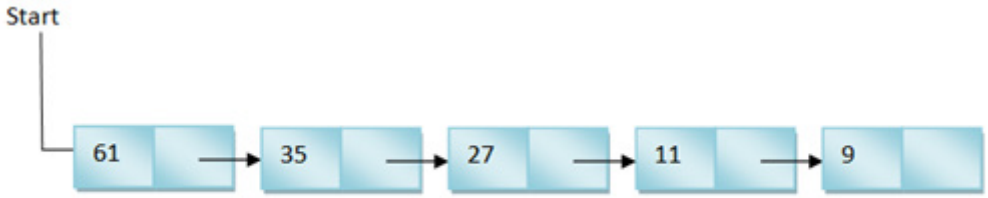
Next, Find the minimum element in the remaining list (61 to 27) and swap it with the second last element (27)



// 11 and 27 are swapped

Now, 9 and 11 are sorted and placed in their right position.

Find the minimum element from the remaining list (from 61 to 35). It is 27. So, swap the element at the third last element.ie,35



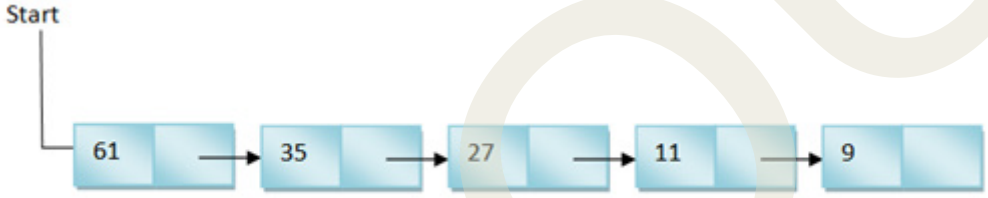
Find the minimum element from the remaining list (from 61 to 35). It is 35. Here, 35 is to be swapped with 35 itself.

61 **35 27 11 9**

Now, only one element remains, and it is in its right position.

61 35 27 11 9

The linked list, after sorting, is



2.2.5 Linked List Vs Arrays

Arrays	Linked List
Arrays are stored in contiguous locations.	Linked lists are not stored in contiguous locations.
Fixed in size	Dynamic in size
Memory is allocated at compile time	Memory is allocated at run time
Uses less memory than a linked list	Uses more memory because it stores both data and the address of the next node
Elements can be accessed easily	Element accessing requires the traversal of the whole linked list
Insertion and deletion operation takes time.	Insertion and deletion operations are faster

Recap

- A node can be inserted at the beginning, end and in a position
- A node can be inserted at the beginning of the list by copying the content of Start into the link part of the new node and the address of the new node into Start.
- To insert a node at the end of the list, we have to copy the address of the new node to the link part of the last node. Assign NULL in the link part of the newly added node.
- A node can be deleted from the beginning, end and in a position
- Searching in a linked list involves traversing through the nodes of the list to find a specific value.
- Sorting a linked list means arranging the elements in an order
- Linked lists are stored in non-contiguous memory locations.
- Linked list is dynamic in size.
- Insertion and deletion operation is faster in linked lists.
- Insertion of an item in a linked list is the process of placing the node containing the item in a particular position

Objective Type Questions

1. Which pointer needs to be updated when inserting a node at the beginning of a singly linked list?
2. What do you need to update when deleting the last node in a singly linked list?
3. What traversal method is used to search for a specific value in a singly linked list?
4. What is the primary pointer that is used to traverse a singly linked list?
5. What is the main difference in memory allocation between a linked list and an array?
6. Which data structure, linked list or array, allows for more efficient insertion and deletion of elements?
7. How does the fixed size of an array compare to the size flexibility of a linked list?

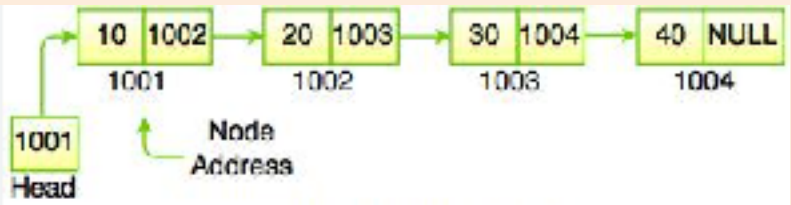
8. What are the applications of linked lists?
9. Which algorithm is not feasible to implement in a linked list?
10. Which data type is used for implementing a linked list in C?

Answers to Objective Type Questions

1. Start/Head
2. Previous node address
3. Linear
4. Start/Head
5. Dynamic
6. Linked list
7. Arrays are fixed-size, linked lists are dynamic-size
8. Implementation of stack, queue, trees
9. Binary search
10. Structure

Assignments

1. Explain how a node is inserted at the end of a linked list.
2. Describe the steps for inserting a node at a specified location in a linked list
3. Discuss the step-by-step procedure for deleting node 30 from the given linked list.



4. Distinguish between Arrays and Linked list
5. Explain how sorting can be done in a linked list

Suggested Reading

1. “Data Structure using C” by A K Sharma.
2. “Data Structures and Program Design in C” by Kruse Robert L.
3. “Data Structures and Algorithms Analysis in C” by Mark Allen Weiss.
4. “Data Structures and Algorithms” by Alfred V Aho and Jeffrey D Ullman.

UNIT 3 Circular Linked List, Doubly Linked List

Learning Outcomes

After completion of this Unit, the learner will be able to:

- To introduce the concept of a Circular Linked list
- To discuss traversal of a Circular Linked list
- To understand the structure of a doubly linked list (DLL)
- To discuss the node insertion process of DLL
- To explain the process involved in DLL node deletion

Prerequisites

Linked lists are a way to represent a list of items. Each element of the list is made to point to the next element in the list, as shown in Figure. An element in the list is called a node. The node is a self-referential structure, having two parts: Data and Next, as shown in Figure. The Data part contains the information about the element, and the Next part includes a pointer to the next element or node in the list.

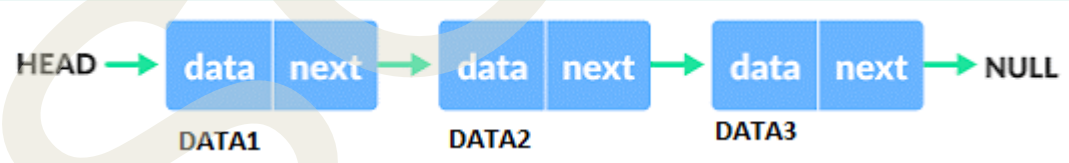


Figure 2.3.1 Linked List

It may be noted that the list is pointed by a pointer called 'ptr'. Currently, the List points to a node containing the data 'Data1', the next part pointing to 'Data2', the next part pointing to 'Data3', and so on. The end of the list is denoted by a pointer called NULL.

A singly linked list allows traversal only in one direction. You can move forward but not backward. Also, while traversing, it is a complex task to keep the address of the previous node. How can we solve all these problems associated with a singly linked list? The answer is a Doubly linked list.

A type of linked list where the last node of the list points back to the first node, forming a circle or loop, is called a Circular linked list.

Key Concepts

Circular Linked list, Doubly Linked list, Node insertion, Node deletion, Node navigation

Discussion

2.3.1 Circular Linked List

In the Singly Linked list, every node points to its next node in the sequence and the last node points to NULL. A circular Linked list is circular, which means it has no end. The last node points to the first node, creating a circle, hence the name.

Media player that repeats endlessly where the last song points to the first song is an example of a Circular linked list. Also, in multiplayer games, all players are placed in a circular linked list.

A circular linked list is similar to the singly linked list except that the last node points to the first node in the list.

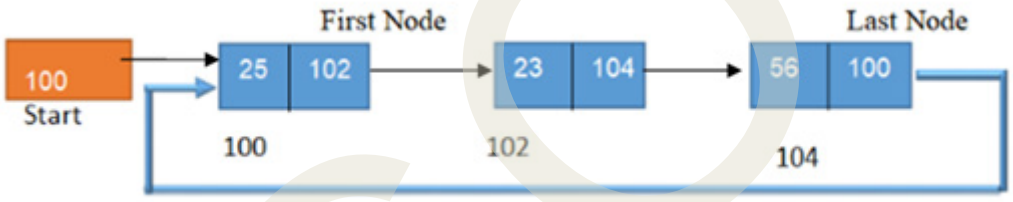


Fig.2.3.2 Circular Linked list

2.3.1.1 Creation of Circular linked list

The algorithm for creating a circular linked list is given below.

Step 1: Take a new node in the pointer called Start

Step 2: Read Data (Start)

Step 3: Take a pointer called End and point it to the same node being pointed by Start,

i.e, Start= End.

Step 4: Bring a new node in the pointer called Temp

Step 5: Read Data (Temp)

Step 6: Connect the link part of End to Temp, i.e., End→link = Temp

Step 7: Set End = Temp

Step 8: Repeat steps 4 to 7 till the whole of the list is constructed

Step 9: Point Link of Temp to First

Step 10: Stop.

2.3.1.2 Traversing in Circular Linked List

The main advantage of a circular linked list is that from any node, one can reach any other node. Traversing in a circular linked list can be done by a loop. Initialise the temporary pointer variable **Temp** to **Start** pointer and run until the next pointer of **Temp** becomes **Start**. The algorithm is described as follows.

Algorithm

Step 1: The address of the first node is found from Start and stored in Temp.

Step 2: If the content of Temp = NULL, Stop

Step 3: else, using the address in Temp, get the data of the first or next node.

Step 4: The content of the link part of this node (i.e. the address of the next node) is stored in Temp

Temp = Temp → Link //value of temp is updated to link of temp.

Step 5: Repeat Step 4 and Step 5 until Temp → Link! = Start

//Till the last node, where the link part points to the start address

Step 6: Stop.

2.3.2 Structure of Doubly Linked List

We studied that a linked list is a collection of nodes connected using a link. Observe the linked list given below. How does it differ from the singly linked list? Each node has two links: the left link and the right link. Or we can say navigation is possible in forward and backward directions. The linked list, in which navigation is possible in both the forward and backward directions, is called a doubly linked list.

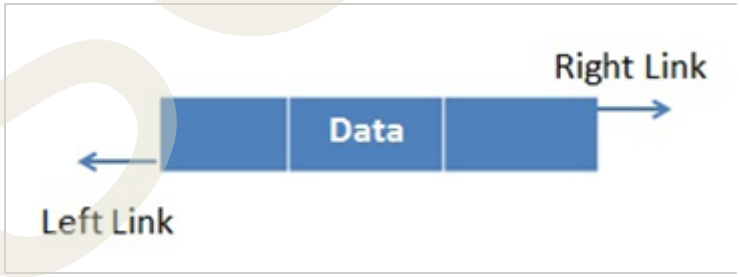


Fig.2.3.3 Node of a doubly linked list

From Fig.2.3.3, it can be noticed that a node in the doubly linked list has the following three fields:

(1) Data: for the storage of information.

(2) Left Link: pointer to the preceding node.

(3) Right Link: pointer to the next node.

The left link of the first node is NULL because there is no preceding node for the first node. Similarly, the right link of the last node is NULL.

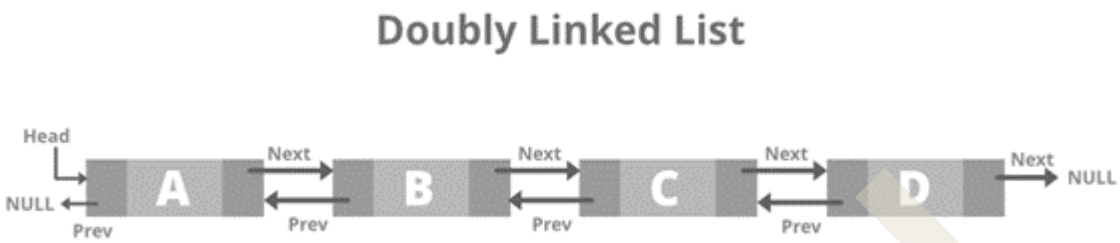


Fig 2.3.4 A Doubly linked list

2.3.3 The Algorithm for the Creation of a Doubly Linked List

The algorithm for the creation of a doubly linked list is given below:

Algorithm

Step 1. Take a new node in the pointer called First.

Step 2. Point leftLink of first to NULL, i.e., first−>leftLink = NULL.

Step 3. Read Data(First).

Step 4. Point back pointer to the node being pointed by First, i.e., back = First.

Step 5. Bring a new node called Far to the pointer.

Step 6. Read Data(Far).

Step 7. Connect rightLink or back to Far, i.e., back −> rightLink = Far.

Step 8. Connect leftLink of Far to Back, i.e., Far −>leftLink = Back.

Step 9. Take back to Far, i.e., back = Far.

Step 10. Repeat steps 5 to 9 till the whole of the list is constructed.

Step 11. Point rightLink of far to NULL, i.e., Far −> rightLink = NULL.

Step 12. Stop.

2.3.3.1 Insertion of a Node in Between Two Nodes in Doubly Linked List

Consider the following doubly linked list, which represents the insertion of a New Node in between two nodes, as shown in Figure 2.3.5

We are given a pointer to a node as **prev node**, and the new node is inserted after the given node. This can be done using the following steps:

First, create a new node (say, new node).

Now insert the data in the new node.

Point the next of the new node to the next of prev node.

Point the next of prev_node to new_node.

Point the previous of new node to prev_node.

Point the previous of next of new node to new_node.

The following program segment can do insertion of New Node:

new_node->data = new_data

new_node->next = prev_node->next

prev_node->next = new_node

new_node->prev = prev_node

if (new_node->next != NULL)

new_node->next->prev = new_node

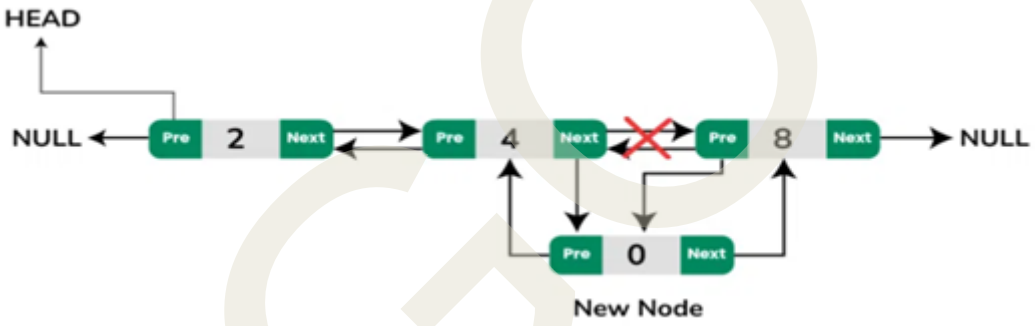


Fig 2.3.5 Insertion of new Node in between two nodes

2.3.4 Deletion of a node in doubly linked list

Consider the following doubly linked list, which represents the deletion of a node at the specified position, as shown in Figure 2.3.6

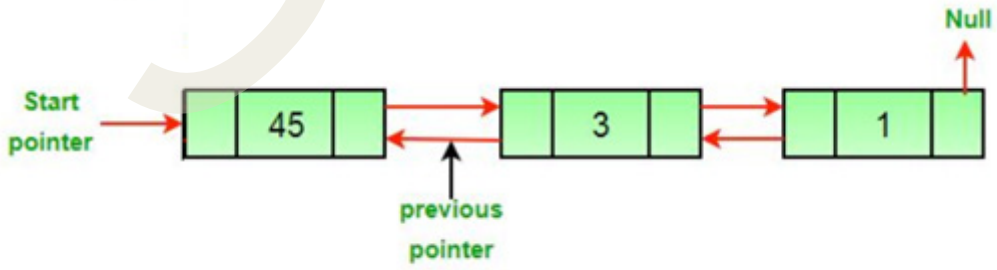


Fig 2.3.6 Deletion of a node in doubly linked list

Algorithm:

- Let the node to be deleted be del.
- If the node to be deleted is head node, then change the head pointer to the next current head.
- Set prev of next to del if next to del exists
- Set next to previous to del if previous to del exists.

The following program segment can do deletion in Doubly Linked List Deletion of a node:

```
(1) if (del->next != NULL)
        del->next->prev = del->prev
2) if (del->prev != NULL)
        del->prev->next = del->next
3) free(del)
```

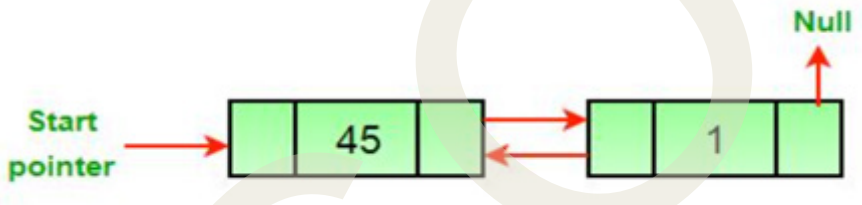
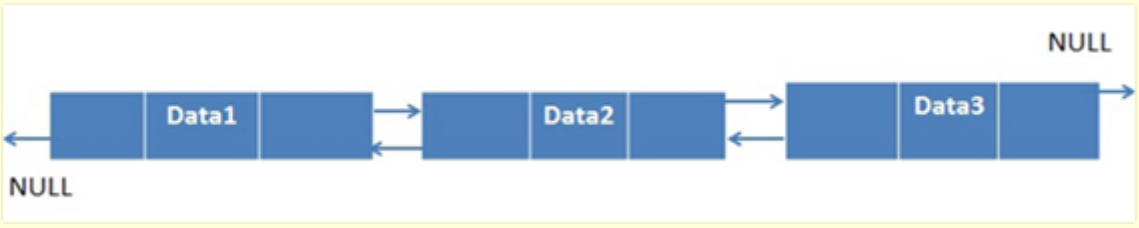


Fig 2.3.7 After the deletion of the middle node

Recap

- In a circular linked list, the last node points back to the first node, forming a circle.
- Traversing a circular linked list can start from any node and continue until it returns to the starting node.
- Unlike singly linked lists, circular linked lists do not have a null pointer to indicate the end of the list.
- A doubly linked list is a two-way list in which all nodes will have two links.



- This helps in accessing both the successor node and predecessor node from the given node position.
- Each node in a doubly linked list contains three fields: Left link, Data and Right link.
- The left link points to the predecessor node, and the right link points to the successor node.
- The data field stores the required data.
- In doubly linked lists other than traversal, insertion and deletion of a node is also possible.
- We can navigate in both directions
- It requires more space than a singly linked list
- The insertion and deletion of a node take a bit longer

Objective Type Questions

1. What differentiates a circular linked list from a singly linked list?
2. Name the data structures that maintain two pointers to store the next and previous nodes.
3. What is the number of pointers affected by an insertion operation in a doubly linked list?
4. Give an application of a circular linked list.
5. Give an application of a doubly linked list.
6. In a circular linked list, what should the pointer to the last node be set to if the list becomes empty?
7. In a circular linked list, which node does the last node point to?

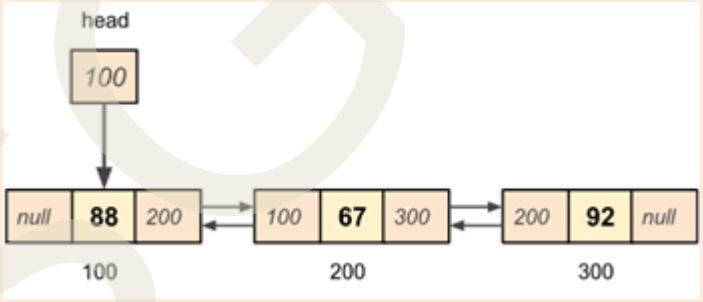
Answers to Objective Type Questions

1. There is no Null pointer
2. Doubly linked list

3. Two (Each node has only one pointer to traverse the list back and forth).
4. Allocating CPU resources.
5. Used in navigation system
6. NULL
7. First

Assignments

1. What is a Circular linked list? Explain the basic operations of a circular linked list.
2. Write a function to detect if a given singly linked list has a loop or cycle. If it does, convert the loop into a circular linked list by ensuring the last node points back to the first node.
3. Explain the algorithm for creating a doubly linked list.
4. Create the following doubly linked list. Write the algorithm for adding a new node at the 2nd position.



Suggested Reading

1. “Data structure using C”, by AK Sharma
2. “Data Structure and Algorithms”, by Alfred V Aho and Jeffrey D Ullman.
3. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss
4. Online resource: TutorialsPoint - Data Structures - Circular Linked List

UNIT 4

Linked List Representation of Stack and Queue

Learning Outcomes

After completion of this unit, the learner will be able to:

- familiarise with linked list realisation of the stack
- introduce the basic stack operations in linked list implementations
- discuss Queue implementation using linked lists.

Prerequisites

Stack

A stack is an ordered collection of homogeneous data elements where the insertion (PUSH) and deletion (POP) of elements are taking place at one end. We can often call it the top of the stack. Stack follows the Last In First Out (LIFO) mechanism. A simple real-world example is a stack of plates, as shown in Fig. 1(a). We can add a new plate on the top of the stack. The last plate placed on top of the stack is the first to be taken off.

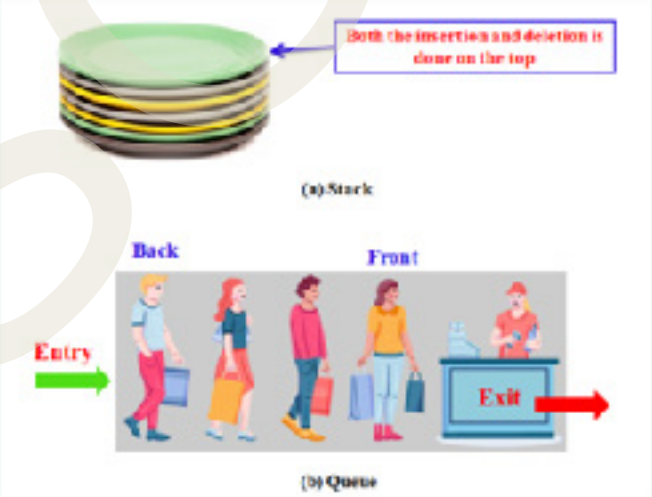


Fig : Real-world examples for Stack and Queue

Queue

A queue is a linear list of elements in which deletion is done at the front end, and insertion

is done at the rear end. That is, the queue follows the First In First Out mechanism. We can simply call it FIFO. A real-world example of a queue is people standing in a queue at the bill counter in a supermarket for payment. Fig. 1 (b) shows an example of a queue. Here, a person enters the queue at the back side and leaves the queue from the front side.

We have already studied the array representation of both Stack and Queue. In this unit, we will discuss their linked list representation.

Key Concepts

Queue, Stack, Linked list

Discussion

2.4.1 Linked List Representation of Stack

The major problem of stack representation using an array structure is that it works only for a fixed number of data items. At the beginning of the implementation, the amount of data must be specified. This representation is not suitable when we don't know the size of the data which we are going to use. The stack can be implemented using a linked list. A stack using a linked list works for variable sizes of data. So, there is no need to fix the size at the beginning of the implementation.

In the linked representation of stacks, the top of the stack is represented by the first item in the list. The first element inserted into the stack is pointed out by the second element inserted; the second element inserted is pointed out by the third element inserted, and so on. The linked list representation of a stack is shown in Fig. 2.4.1. The top always points to the last item inserted.

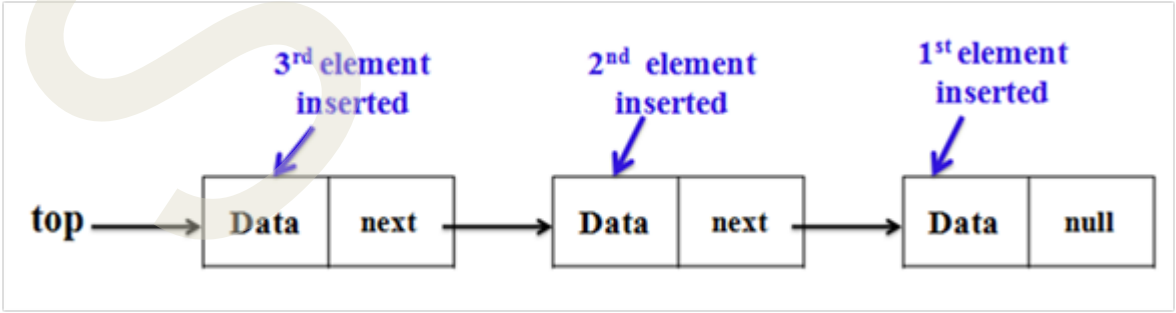


Fig 2.4.1 Linked list representation of stack

In this representation, both the insertion and deletion operations do not involve more data movements. When the user wants to push an element into the stack, then only the memory space is allocated. So the memory space is not wasted. To implement a push, we create a new node in the list and attach it as the new last element inserted (the first

node in the list). To implement a pop, we advance the top of the stack to the second item (node) in the list.

2.4.1.1 Push Operation

To insert an element into the stack, we use the PUSH operation. The following steps involve pushing an element onto the stack.

1. Create a node first.

2. If the list is empty, then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assigning null to the address part of the node.

3. If there are some nodes in the list already, then we have to add the new element at the beginning of the list. For this purpose, assign the address of the starting element to the address field of the new node and make the new node the starting node of the list.

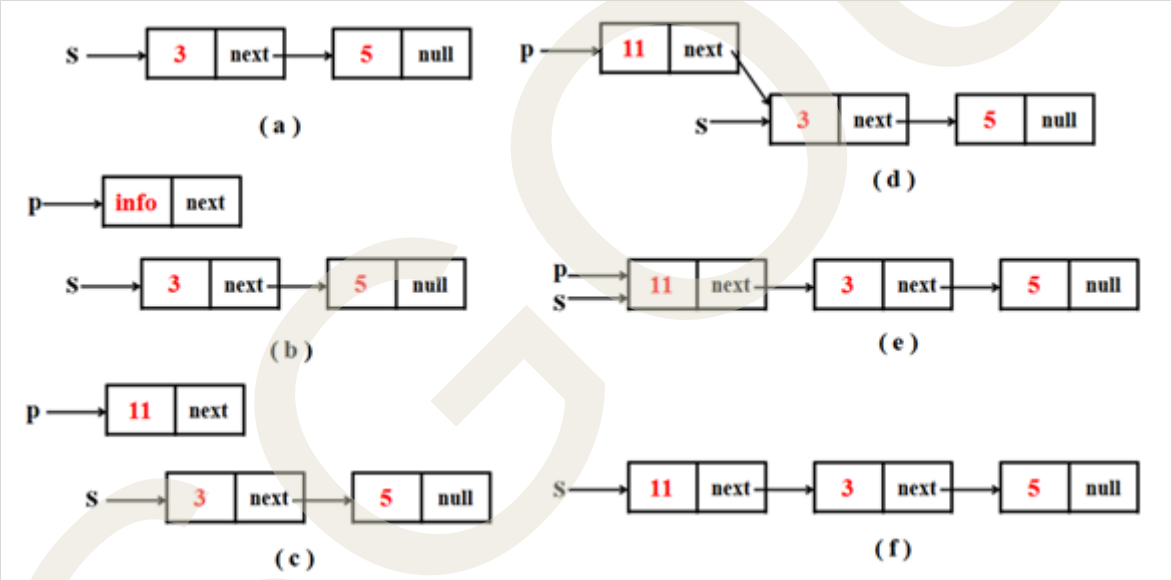


Fig 2.4.2 PUSH Operation using linked list

For example, let's consider a list shown in Fig.2.4.2(a). Here, two elements are there: 3 and 5. The top component of the stack is 3. Now, we are adding a new item, 11, to the list. To insert a new element 11, we have to create a new node. Fig.2.4.2(b) shows the result of this operation. We then assign the value "11" to the info part of the node p. So 11 is stored in the data part of the new node p. The result of this operation is shown in Fig 2.4.2(c).

Then, we assign the address of the first node on the list to the next field of node(p). Fig 2.4.2(d) illustrates the result of this operation. Then, p points to the list with the additional items included. Since "s" is the **top** of the desired list, its value must be modified to the address of the new first node of the list. This can be done by s = p. This changes the value of "s" to the value of "p". The result of this operation is shown in

Fig.2.4.2(e). Fig.2.4.2(f) shows the final list after the push operation

2.4.1.2 Pop Operation

Deleting a node from the top of the stack is referred to as a POP operation.

First, we check if the stack is empty or not. If the stack is empty, then we can't perform a pop operation. If the stack is not empty, we can perform a pop operation. Since the elements are inserted at the top of the stack, they are also deleted from the top. Consider the following linked Stack with four nodes. In order to POP the top element, modify the value of Top with the address stored in the link field of the first element.

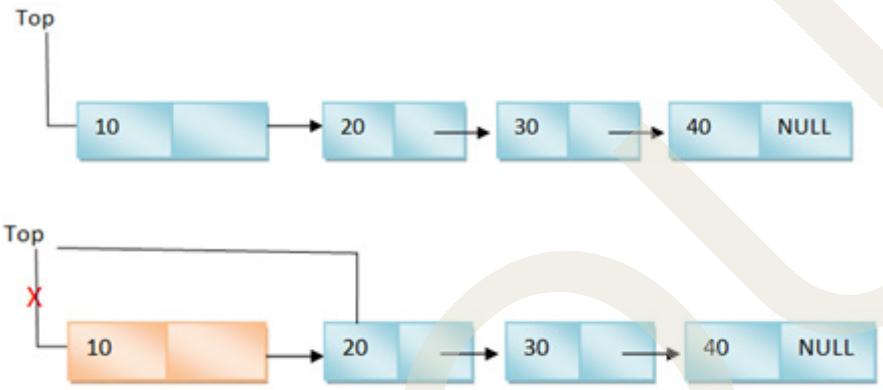


Fig 2.4.3 POP Operation using linked list

Steps for POP operation

1. Check whether the stack is empty. If the top pointer is NULL, raise an error indicating that the stack is empty.
2. Store the top node in a temporary variable.
3. Set the top pointer to the next node of the current top node.
4. Return the value of the popped node.

2.4.2 Linked List Representation of Queue

Queues can be represented by using linked lists. In the queue data structure, the items are inserted at the rear of the queue and deleted from the front of the queue. Let a pointer to the first element of a list represent the front of the queue. Another pointer to the last element of the list represents the rear end of the queue, as shown in Fig. In linked representation, a queue q consists of a list and two pointers, front and rear. In an empty queue, both the front and rear must be null.

2.4.2.1 Insertion or Enqueue

Firstly, check whether the queue is empty. If empty, insert the new node as the first element by giving the address of the new node to both the front and rear. Otherwise, set the link field of the last node with the address of the new node and set the rear pointer

with the address of the new node. Thus, the element gets inserted at the end of the queue.

Algorithm

```
p = getnode( );
info(p) = x;
next(p) = null;
if (rear = = null)
{
front = p;
}
else
{
next(rear) = p;
rear = p;
}
```

For example, let's consider a list shown in Fig.2.4.4(a). Here are two elements: 5 and 7. The front element (front) is five, and the rear element (rear) is 7. Now, we are adding a new item, 13, to the list. For inserting a new element 13, we have to create a new node p. Here p = getnode() creates a new node "p". We then assign the value "x" to the info part of the node p. Here the value of "x" is 13. So 13 is stored in the data part of the new node p.

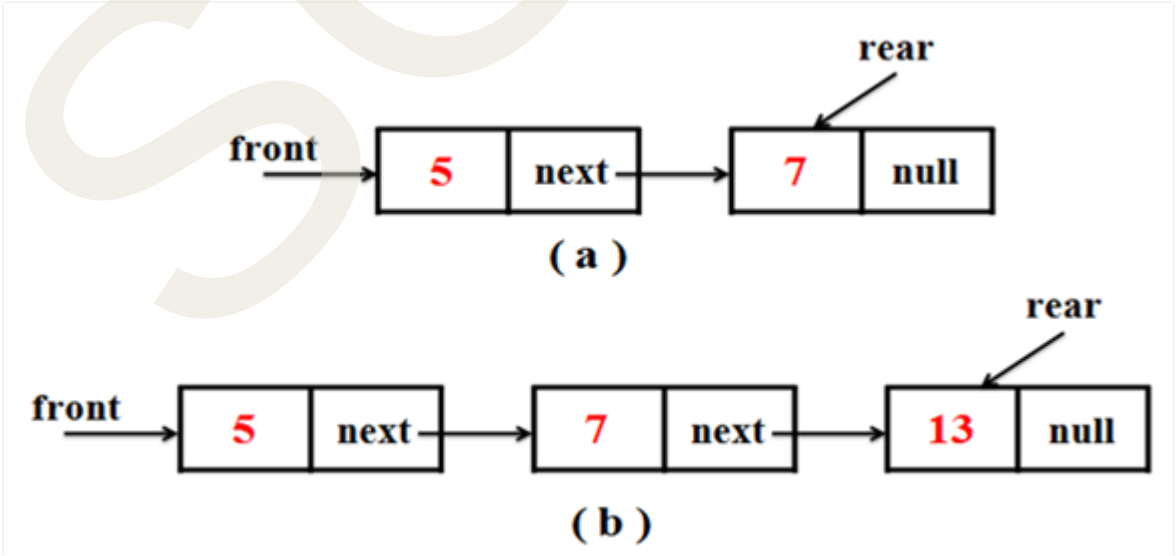


Fig 2.4.4 Insertion using linked list

We all know that the insertion in a queue takes place at the rear end. So we assign “null” to the next(p). That is, this operation places “null” into the next field of node(p). Then, we check whether the rear of the queue is “null” or not. If “null”, then place p into front. Otherwise, place p into next(rear). Finally, assign p to q.rear. Now, p (node with value 13) is the rear element in the queue. Fig.2.4.4(b) shows the list after inserting the element 13.

2.4.2.2 Deletion or Dequeue

The delete operation in a queue removes the element that was first inserted, which is located at the position indicated by FRONT. Before performing the deletion, it's essential to check whether FRONT is equal to NULL. If FRONT is NULL, it means the queue is empty, and no further deletions can be carried out. When trying to delete a value from an empty queue, an underflow message is displayed.

Algorithm to delete an element from a linked queue

Step 1: IF FRONT = NULL

Write Underflow

Go to Step 5

[END OF IF]

Step 2: SET PTR = FRONT

Step 3: SET FRONT = FRONT -> NEXT

Step 4: FREE PTR

Step 5: END

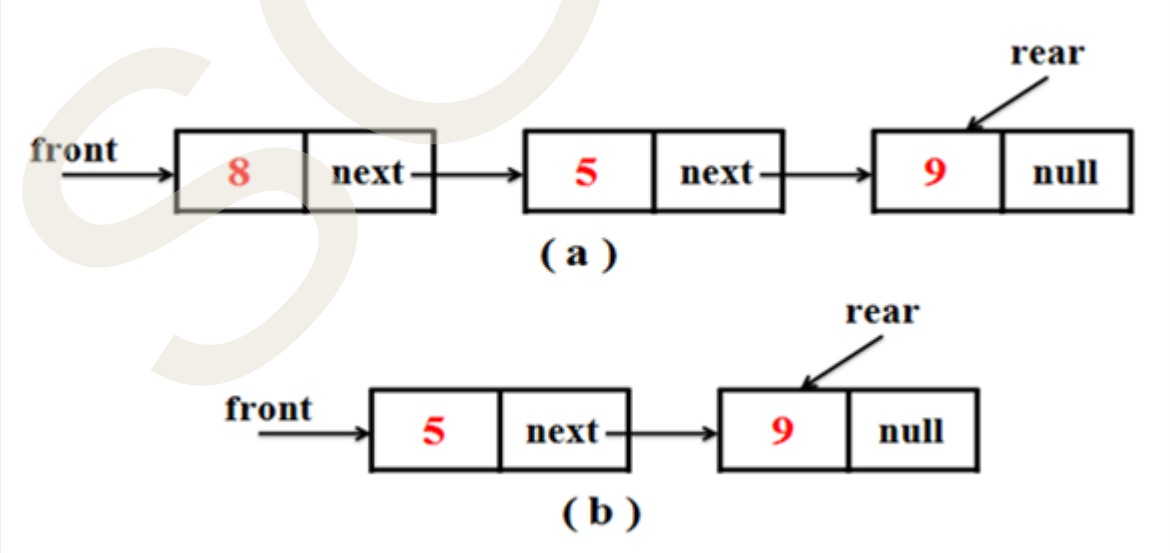


Fig 2.4.5 (a) Linked queue contains three nodes (b) Final linked queue after deletion of node 8

Step 1: Check if FRONT is NULL. As you can see from Figure 2.4.5 (a), the FRONT is not NULL, and it points to Node 5, so we proceed to Step 2.

Step 2: Set PTR to the node currently at FRONT of the linked queue. It is clear from the figure that the node currently at FRONT is node 8. So, PTR now points to the node 8.

Step 3: The next step is to update FRONT to point to the next node. And FRONT is now set to FRONT -> NEXT, which is Node 5.

Step 4: Free the node pointed to by PTR. PTR points to the node 8. So, Node 8 is freed from memory.

The final list after deletion of element 8 is shown in Fig.2.4.5(b).

Recap

- A stack using a linked list works for variable sizes of data.
- In the linked representation of stacks, the top of the stack is represented by the first item in the list.
- The first element inserted into the stack is pointed out by the second element inserted; the second element inserted is pointed out by the third element inserted, and so on.
- The top always points to the last item inserted.
- To push, create a new node and attach it as the new first node.
- To pop, move the top to the second item in the list.
- Queues can be represented by using linked lists.
- In a queue, items are inserted at the rear and deleted from the front.
- In an empty queue, both front and rear must be null.

Objective Type Questions

1. A stack using a linked list works for variable size of data. True or False?
2. Which node in the linked representation of stacks represents the top of the stack in the list?
3. What operation is used for inserting an element into the stack?

4. What does the operation x = pop(s) remove from a non-empty list?
5. What operation is to be performed for removing an item "x" from the queue "q"?
6. In the queue, the items are inserted at the rear of the queue and deleted from the front of the queue. True or False?
7. In an empty queue, both q.front and q.rear must be null. True or False?

Answers to Objective Type Questions

1. True
2. The first item or node
3. push
4. the first node
5. front, rear
6. True
7. True

Assignments

1. Write a C program to implement a stack using linked lists.
2. Write a C program to implement a queue using linked lists.
3. What are the advantages of representing a group of items as an array versus a linear linked list?
4. What are the disadvantages of representing a group of items as an array versus a linear linked list?
5. Explain the linked list representation of stacks.
6. Explain about the representation of the queue using a linked list.
7. Describe the push operation in stacks using linked lists.

8. Describe the pop operation in stacks using linked lists.

9. How is insertion and deletion done in a queue using a linked list representation? Explain.

Suggested Reading

1. "Data Structures using C and C++" by Y. Langsam, Moshe J. A. and Aaron M. Tenenbaum.

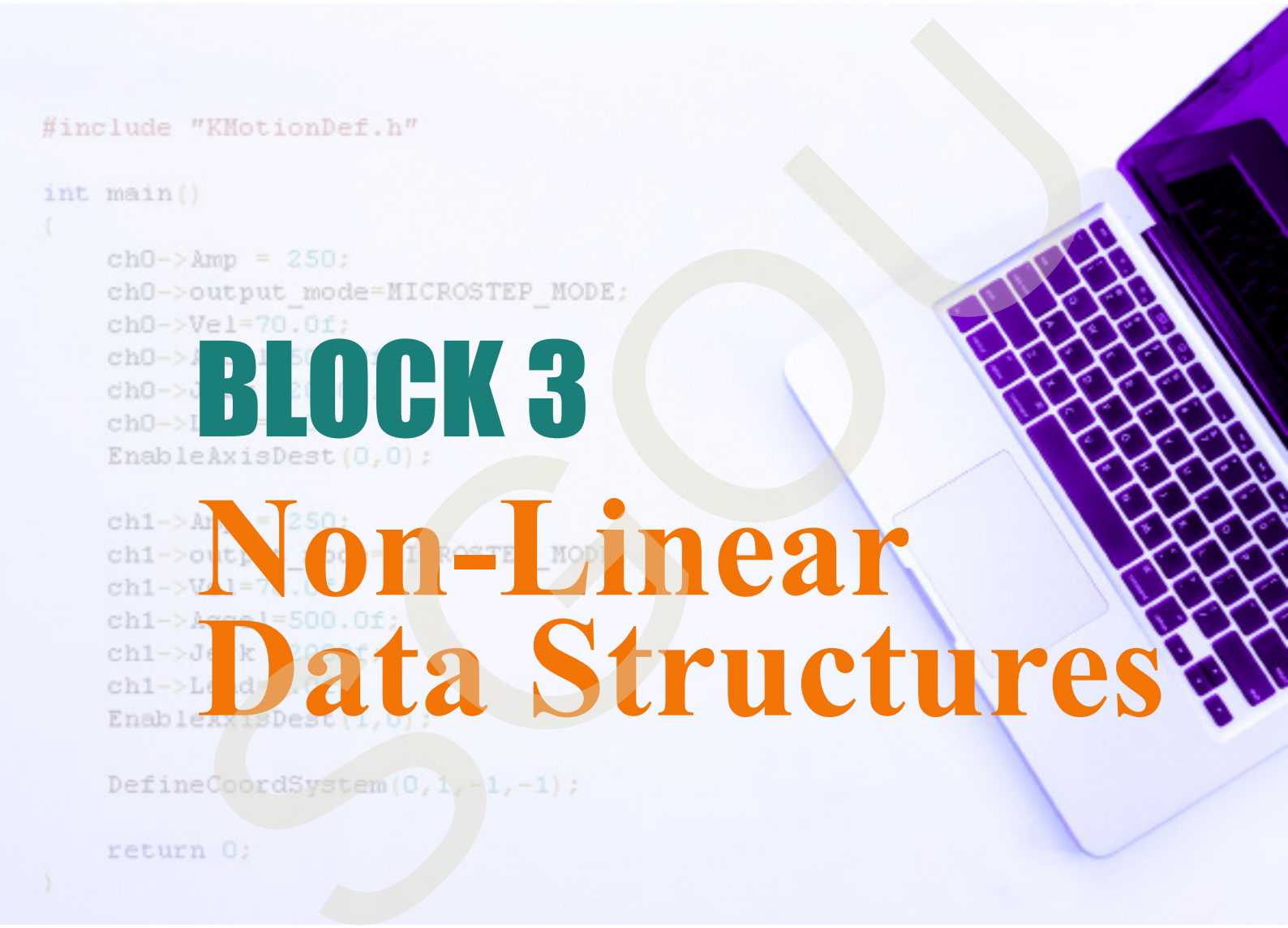
2. "Data Structures Using C" by A K Sharma.

3. "Data Structures Using C" by Udit Agrawal.

4. "Data Structures and Program Design in C" by Kruse Robert L.

5. "Data Structures and Algorithms" by Alfred V Aho and Jeffrey D Ullman.

6. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.



BLOCK 3

Non-Linear Data Structures

UNIT 1 Trees

Learning Outcomes

Upon completion of this Unit, the learner will be able to:

- gain awareness of the different types of non-linear data structures
- familiarise with the concepts and terminologies of tree data structure
- study the concept of a binary tree and important types of binary trees, such as complete binary trees, full binary trees, and 2-tree or extended binary trees
- study the concepts of tree traversal algorithms, such as inorder traversal, preorder traversal and postorder traversal

Prerequisites

In the previous units, we discussed arrays, stacks, queues, and linked lists, which are linear data structures. If we require ordered or sequential information, we use these linear data structures.

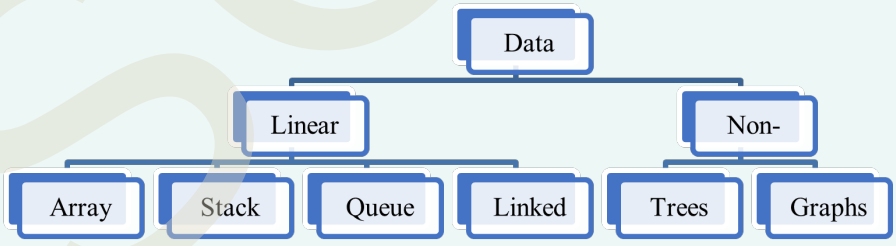


Fig 3.1.1(a) Classification of Data Structures

In the linear data structure, data is stored sequentially. In this structure, every element has a unique predecessor and unique successor. In Figure 3.1.1(b), we can see a linear data structure. Here, 1, 2, 3, 4 and 5 are stored in consecutive memory locations. The successor of 1 is 2, and the successor of 2 is 3 and so on. The predecessor of 2 is 1, and the predecessor of 3 is 2 and so on. Examples of these types of data structures are arrays, linked lists, stacks and queues.

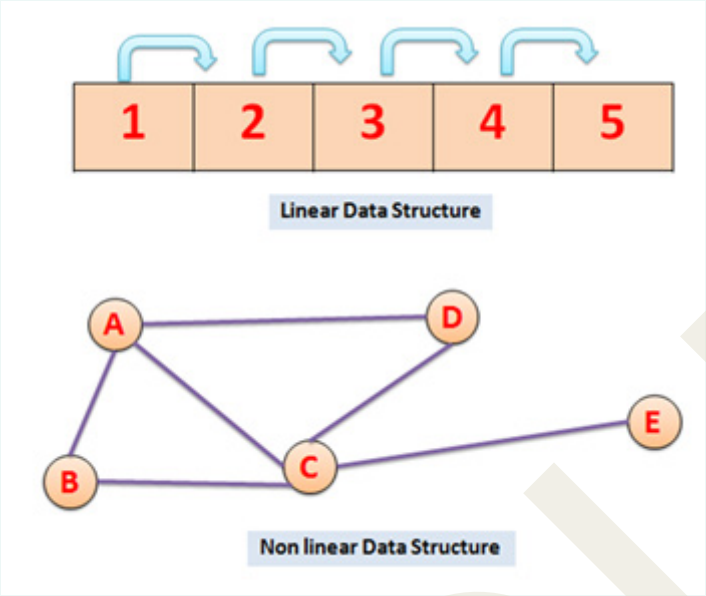


Fig 3.1.1(b) Types of Data Structure

In a non-linear data structure, the data is stored in a distributed manner. That is, the data is stored in non-sequential form. So, there is no single previous or next element. Elements in the non-linear data structure do not form a sequence. There is no unique predecessor or unique successor. In Figure 3.1.1(b), we can see a non-linear data structure. Here, A, B, C, D and E are stored in non-consecutive memory locations and do not form a sequence. Examples: trees, graphs, etc.

Trees and graphs come under non-linear data structures. Before studying tree data structures, binary trees, and tree traversal, students should understand basic data structures such as arrays, linked lists, stacks, and queues. They should be familiar with concepts of nodes, pointers, and dynamic memory allocation. A good grasp of recursion and basic algorithmic problem-solving techniques is also essential. Additionally, learners should have basic programming skills to implement and manipulate these data structures effectively.

Key Concepts

Non-linear data structure, Trees, Binary Trees, Complete binary tree, 2-tree or Extended binary trees. inorder traversal, preorder traversal, postorder traversal

Discussion

3.1.1 Tree - Concepts and Terminologies

Trees are hierarchical data structures consisting of nodes connected by edges. Each tree has a root node, which serves as the starting point, and every other node is connected by edges, forming a parent-child relationship. Key terminologies include root, leaf, sibling, and subtree, which help in understanding the structure and relationships within a tree. The following Figure 3.1.2 shows a tree structure. Each line connecting two nodes denotes a relation, namely the parent-child relation between two nodes.

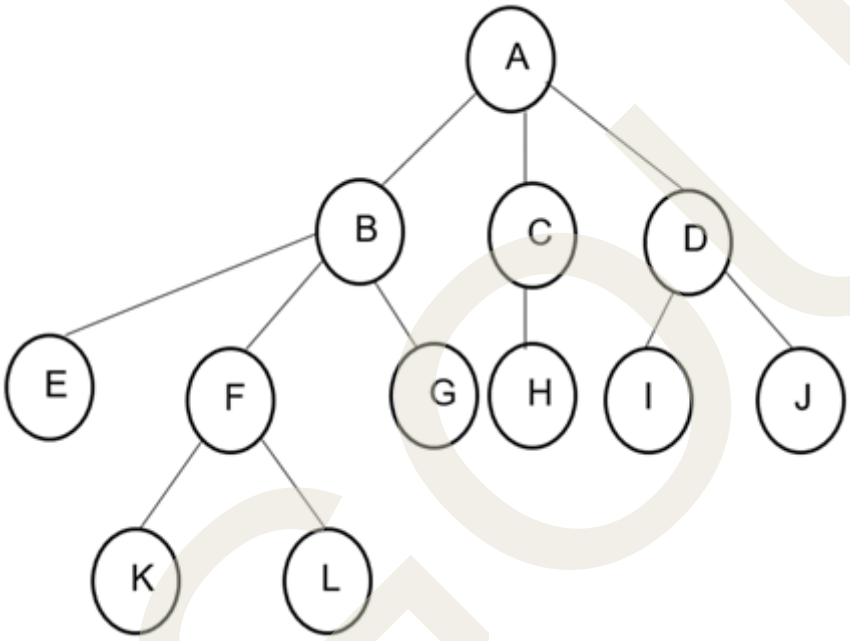


Fig 3.1.2 Structure of Tree

When a person is travelling from Cochin to Mumbai, there are many modes of transportation that he can take. This includes by sea, by road, by air, and by rail. Figure 3.1.3 shows the options available for travel from Cochin to Mumbai.

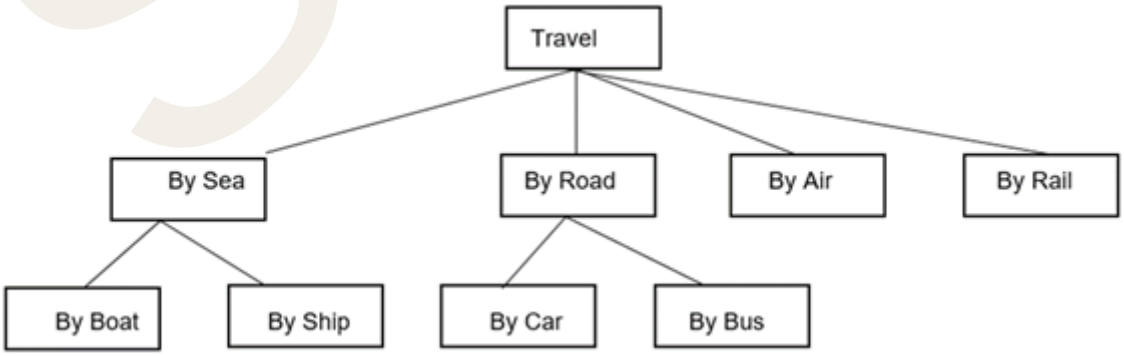


Fig 3.1.3 Options available for travel from Cochin to Mumbai.

When a person is travelling by sea, he can use either a boat or a ship. Similarly, while travelling on the road, he can use either a car or a bus. The node below a given node connected by its edge downward is called its child node. 'By Sea' and 'By Road', 'By Air' and 'By Rail' in Figure 3.1.3 are called child nodes of 'Travel', the root node. A leaf node is a node without any children (For example, By ship, By Boat, By car, By Bus, By Air, By Rail). The root of the tree is the node with no parents. In Figure 3.1.3, travel is the root node.

We can use another example for the tree data structure, i.e., a computer stores information about files and folders in a hierarchical manner in the memory. Figure 3.1.4 shows the root node "course" with subfolders named Mechanical, Computer, Electrical, and Electronics. The subfolders of the Computer are B. Tech and BCA.

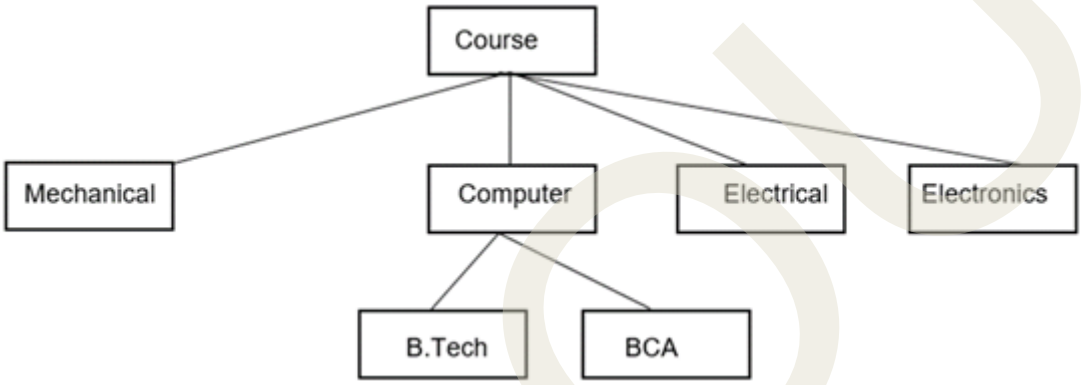


Fig 3.1.4 Example of file and folder hierarchy inside computer

3.1.1.1 Tree - Key Terminologies

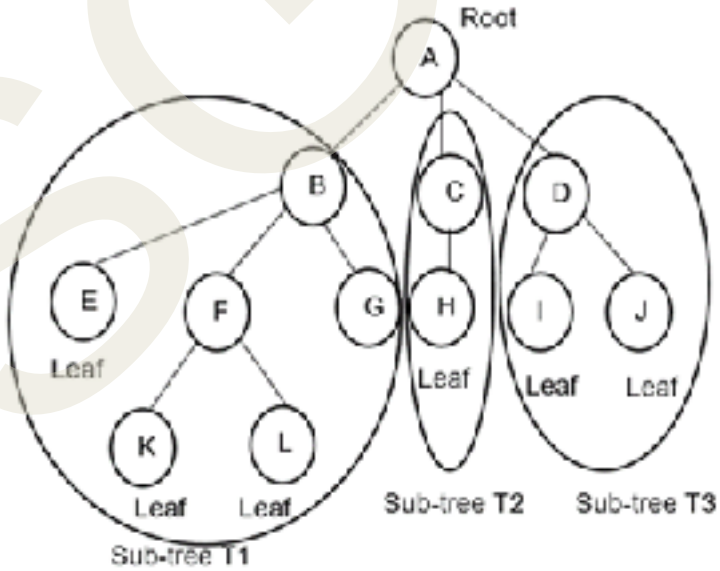


Fig 3.1.5 The Tree and its related terms

From Figure 3.1.5, the following components of a tree can be identified.

- Root: A
- Child nodes of A: B, C, D
- Leaf nodes: E, G, H, I, J, K, L

Now let us discuss a few terminologies associated with a tree:

- **Node -** A tree is a collection of entities called nodes. Nodes are connected by edges. Each node contains a value or data, and it may or may not have a child node. Example: A, B, C…L in the figure 3.1.5.
- **Root -** The node which has no parent. Example: A is the root of the tree given in the figure 3.1.5.
- **Leaf -** The node which has no children is called a leaf or terminal node. Example: I, J, K, L, etc. are leaf nodes in the figure 3.1.5.
- **Height -** The number of nodes present in the longest path of the tree from root to a leaf node is called the height of the tree. This will contain a maximum number of nodes. Example: One of the longest paths in the tree shown in Figure 3.1.5 is A-B-F-K. Therefore, the height of the tree is 4.
- **Depth -** The depth of a node is the length of its path from the root node. The depth of the root node is taken as zero. Example: The depths of nodes G and L are 2 and 3, respectively.
- **Degree -** The degree of a node is defined as the number of children present in a node. The degree of a tree is defined as equal to the degree of a node with maximum children. Example: Degrees of nodes C and D are 1 and 2, respectively. The degree of the tree is 3 as there are two nodes, A and B, having a maximum degree equal to 3. (see Fig 3.1.5)
- **Siblings:** Siblings are the nodes that share the same parent; that is, they are the nodes that are on the same level and directly connected to the same parent node.

3.1.2 Binary Trees

In a normal tree, each node can have any number of children. However, in a binary tree, each node can have at most two children, or it can be an empty tree. A binary tree is a finite set of nodes that either has no nodes or consists of a root node and two separate binary trees known as the left subtree and the right subtree. Figure 3.1.6 illustrates the generic structure of a binary tree.

A non-empty binary tree consists of the following:

- A node called the root node
- A left sub-tree
- A right subtree

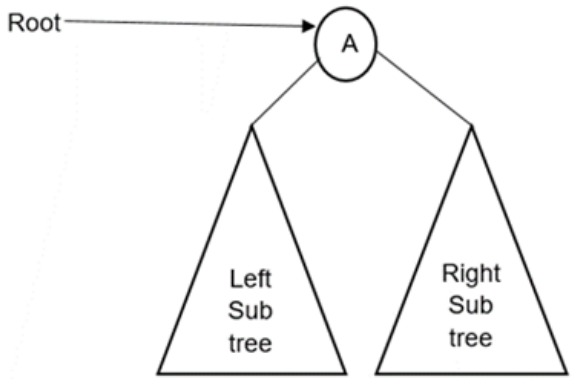


Fig 3.1.6 Generic Binary Tree

Binary trees are classified into five types :

1. Complete Binary Tree
2. Full Binary Tree
3. Perfect Binary Tree
4. Balanced Binary Tree
5. 2-tree or Extended binary trees

3.1.2.1 Complete Binary Trees

In the case of a complete binary tree, all the nodes are filled except the last one. The nodes must be as left as possible in the previous level. The following Figure 3.1.7 is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

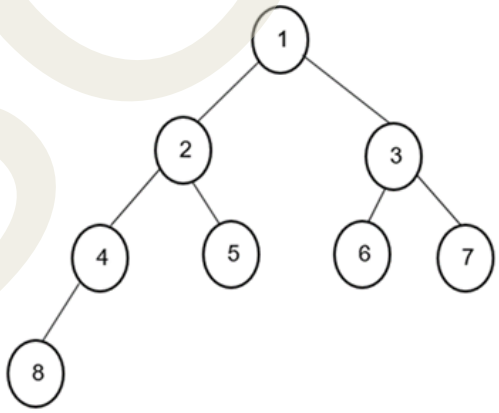


Fig 3.1.7 Complete Binary Tree

3.1.2.2 Full Binary Trees

In a full binary tree, every node other than the leaves has exactly two children, one on the left and one on the right, and each leaf node has no children. This means that each node has exactly zero or two children, as shown in the following Figure 3.1.8.

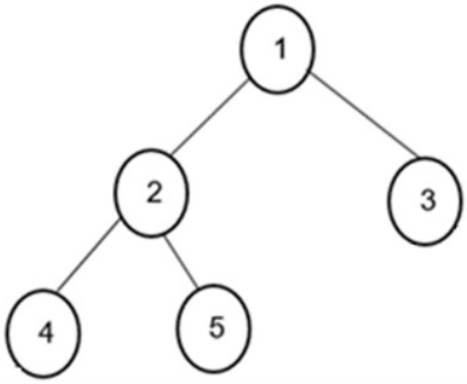


Fig 3.1.8 Full Binary Trees

A complete binary tree is a full tree except at the last level, where all nodes must appear as far left as possible.

3.1.2.3 Perfect Binary Tree

In a perfect binary tree, all Internal nodes have two children, and all leaves are at the same level.

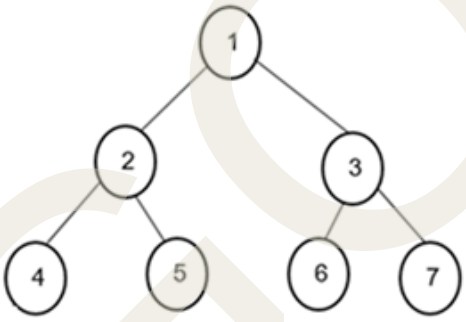


Fig 3.1.9 Perfect binary tree

3.1.2.4 Balanced Binary Tree

The height of the left and right subtrees of any node differs by atmost one.

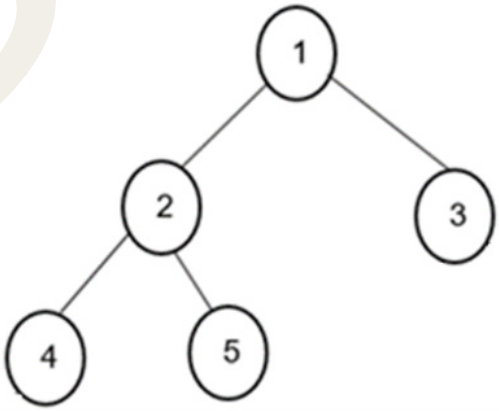


Fig 3.1.10 (a) Balanced Binary Tree

Figure 3.1.10(a) is a balanced binary tree; the height of the left and right binary trees differs by one.

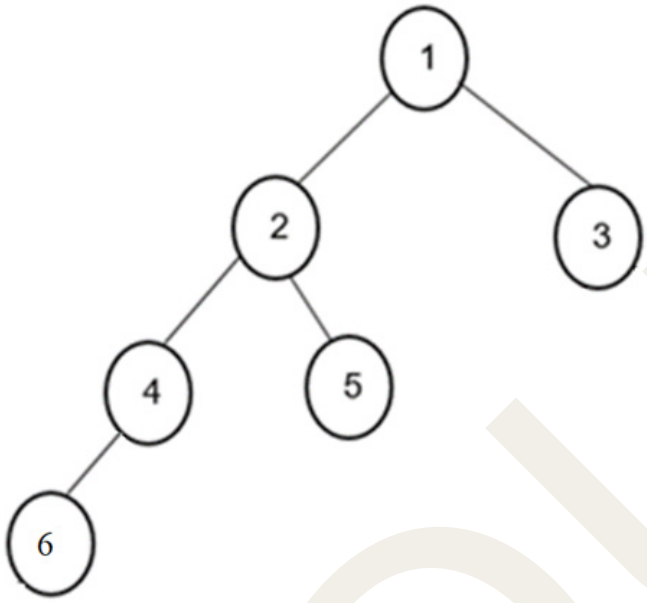


Fig 3.1.10 (b) Unbalanced Binary Tree

Figure 3.1.10(b) is an unbalanced binary tree; the height of the left and right binary trees differs by two.

3.1.2.5 2-tree or Extended binary trees

It is a binary tree with the property that each node has either 0 children or 2 children. Figure 3.1.11 below is an example of extended binary trees. Here, all the nodes except G have two children. G has no children.

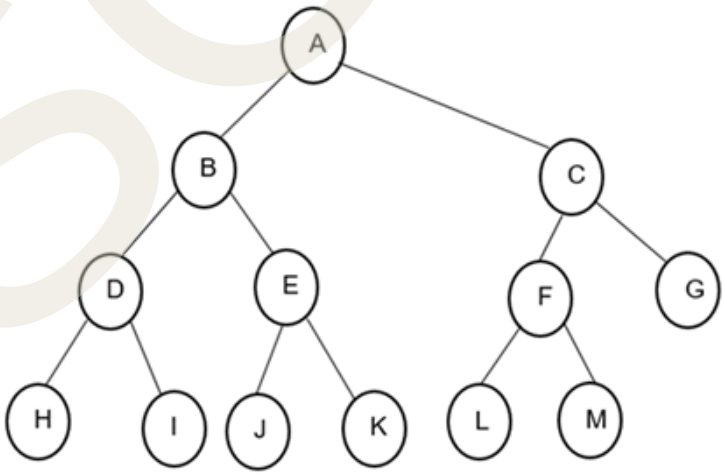


Fig 3.1.11 Extended Binary Tree

3.1.3 Structure of a Node

Each node comprises the following:

1. It contains some information.
2. It has an edge to a left child node.
3. It has an edge to the right child node

The above components of a node can be comfortably represented by a linked list, as shown in Figure 3.1.12. Thus, a node consists of:

a. pointer that points towards the right node (Right Child Address)

b. pointer that points towards the left node (Left Child Address)

c. data element.

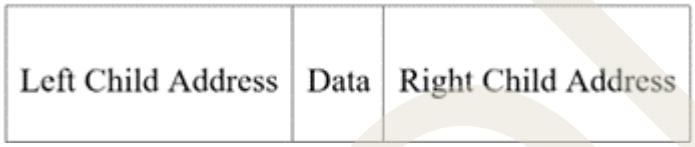


Fig 3.1.12 Node of a binary Tree

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

In a tree, all nodes share a common construct.

3.1.4 Tree Traversal

Suppose you have been given the task of doing system maintenance at three offices in different destinations, namely Cochin, Kozhikode, and Thiruvananthapuram. You can do the maintenance in these cities in the following order.

1. Kozhikode, Cochin, Thiruvananthapuram
2. Kozhikode, Thiruvananthapuram, Cochin
3. Cochin, Thiruvananthapuram, Kozhikode
4. Cochin, Kozhikode, Thiruvananthapuram
5. Thiruvananthapuram, Cochin, Kozhikode

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

6. Thiruvananthapuram, Kozhikode, Cochin

Fig 3.1.13 Kerala Map

If you have a close look at the Kerala Map in Figure 3.1.13, you will realise that Kozhikode is in the northern region of Kerala, while Cochin and Thiruvananthapuram are in the southern region. Moreover, Cochin and Thiruvananthapuram are closer to each other than Kozhikode. Let us take Thiruvananthapuram as the nodal position, Cochin as the left and Kozhikode as the right travel destination. When we set a condition that the left would be visited before the right, then the following three combinations are possible.

A. Cochin, Thiruvananthapuram, Kozhikode

B. Thiruvananthapuram, Cochin, Kozhikode

C. Cochin, Kozhikode, Thiruvananthapuram

In the same manner, a binary tree is traversed for many purposes, such as searching a particular node, processing some or all nodes of the tree, and the like. (In the case of a binary tree, every node can have a maximum of 2 children.) There are three types of tree traversals. Option A in the above example is similar to inorder traversal where left subtree traversal is followed by root followed by right subtree traversal . Option B in the above example is similar to a preorder traversal where the root followed by the left and right subtree is traversed. Option C in the above example is similar to postorder traversal where left subtree traversal is followed by right subtree traversal, followed by root visit.

During tree traversal, all the nodes of a tree are visited, and their values may be printed.

The reasons for binary tree traversal include searching a particular node, processing some or all nodes, etc. The following operations are possible on a node of a binary tree:

1. Process the visited node – V.
2. Visit its left child – L.
3. Visit its right child – R.

Any combination of the above three operations can be done for tree traversal. The tree traversals have been named as preorder, inorder, and postorder according to the operation visit node (V).

3.1.4.1 Inorder Traversal

In the example given in the prerequisite, Cochin, Thiruvananthapuram, and Kozhikode's travel path is similar to inorder traversal, where left subtree traversal is followed by root followed by right subtree traversal. Let us go into the details of inorder traversal.

L-V-R: Inorder traversal, that is, travel the left sub-tree (L), process the visited node (V) and travel the right subtree (R).

Algorithm: An algorithm for inorder travel (L-V-R) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```
inorderTravel(Tree){
if (Tree == NULL) return
else
{
inorderTravel (leftChild (Tree));
process DATA (Tree);
inorderTravel (rightChild (Tree));
}
}
```

Explanation: In the algorithm, an if else statement is included in the beginning to check

If the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Traverse the left subtree, i.e., call inorderTravel (leftChild (Tree));
2. Visit the root i.e.; process DATA (Tree);
3. Traverse the right subtree, i.e., call inorderTravel (rightChild (Tree));

Illustration: Figure 3.1.15 shows the illustration of the Inorder traversal of the binary tree in Figure 3.1.14

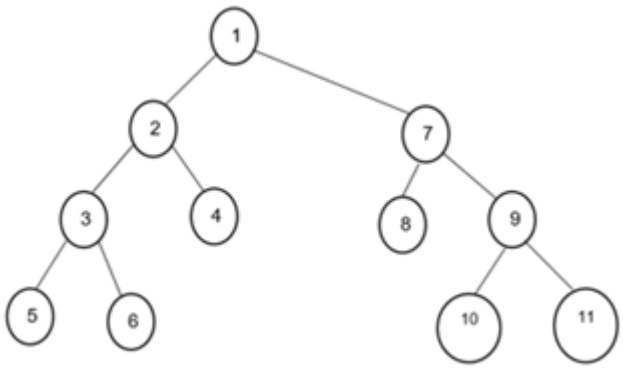


Fig 3.1.14 Binary Tree

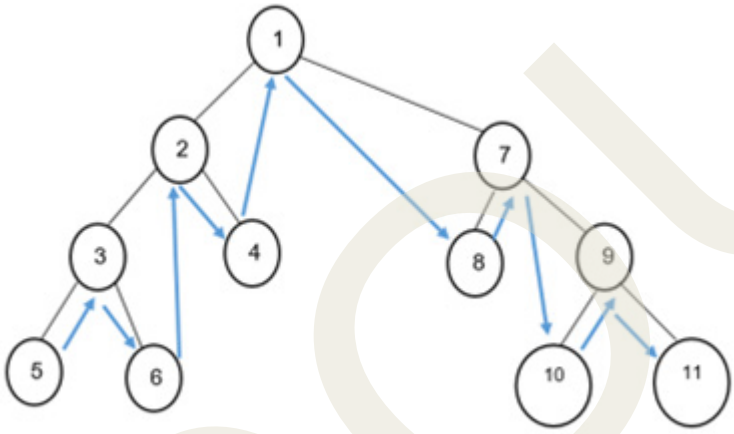


Fig 3.1.15 Inorder traversal

In the case of Figure 3.1.15, the Inorder traversal will give the following result.

5 3 6 2 4 1 8 7 10 9 11

Explanation: In the case of Inorder traversal, L is followed by V, which is followed by R. So, in Figure 3.1.15, the first 5(L) is visited, followed by 3(V) and then 6(R). This is followed by 2(V), which is followed by 4(R). Then 1(V) is visited, followed by 8(L) and then 7(V). This is followed by 10(L), 9(V) and 11(R).

3.1.4.2 Preorder Traversal

In the travel path example discussed above, path B (Thiruvananthapuram, Cochin, Kozhikode) is similar to a preorder traversal where the root followed by the left and right subtree is traversed. Let us now go into the details of preorder traversal.

V-L-R: Preorder traversal, that is, process the visited node (V), travel the left sub-tree (L) and travel the right subtree (R)

Algorithm: An algorithm for preorder travel (V-L-R) is given below. It is provided with a pointer called Tree that points to the root of the binary tree.

preorderTravel(Tree){

```
if (Tree == NULL) return
else
{
process DATA (Tree);
preorderTravel (leftChild (Tree));
preorderTravel (rightChild (Tree));
}
}
```

Explanation: In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Visit the root i.e; process DATA (Tree);
2. Traverse the left subtree, i.e., call preorderTravel (leftChild (Tree));
3. Traverse the right subtree, i.e., call preorderTravel (rightChild (Tree));

Illustration:

In the case of Figure 3.1.16, the Preorder traversal will give the following result.

1 2 3 5 6 4 7 8 9 10 11

Explanation: In the case of preorder traversal, node(V) is visited, followed by L and R. So, in Figure 3.1.16, 1(V) is visited, followed by 2(L), 3(L), and 5(L). Then, this is followed by 6 (R) and 4(R). This is followed by 7(V), which is followed by 8(L). This is followed by 9 (V), 10(L), and 11(R).

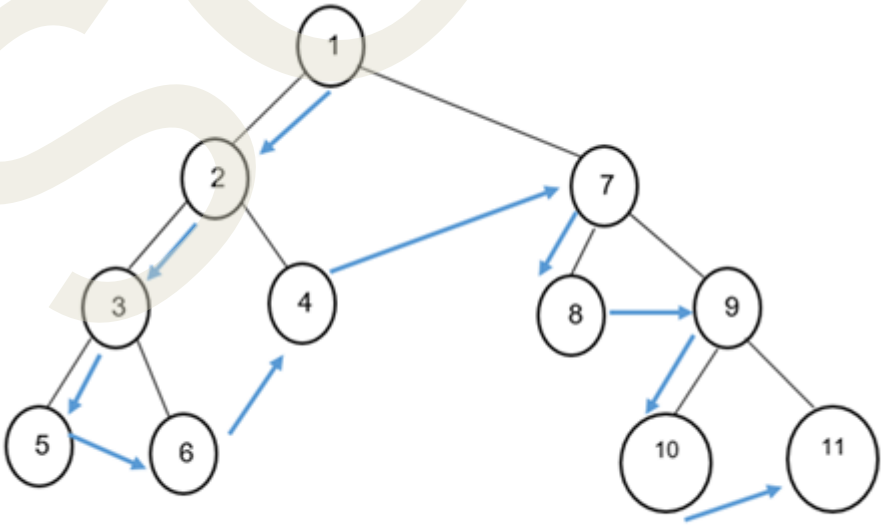


Fig 3.1.16 Preorder traversal

3.1.4.3 Postorder Traversal

In the travel path example, the path C - Cochin, Kozhikode, Thiruvananthapuram travel path is similar to postorder traversal where left subtree traversal is followed by right subtree traversal that is followed by root visit.

Now, let us go into the details of postorder traversal.

L-R-V: Postorder traversal, that is, travel the left sub-tree (L), travel the right sub-tree (R) and process the visited node (V)

Algorithm: An algorithm for postorder travel (L-R-V) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```
PostOrderTravel(Tree){
if (Tree == NULL) return
else
{
postOrderTravel (leftChild (Tree));
postOrderTravel (rightChild (Tree));
process DATA (Tree);
}
}
```

Explanation: In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Traverse the left subtree, i.e., call postorderTravel (leftChild (Tree));
2. Traverse the right subtree, i.e., call postorderTravel (rightChild (Tree));
3. Visit the root i.e; process DATA (Tree);

Illustration :

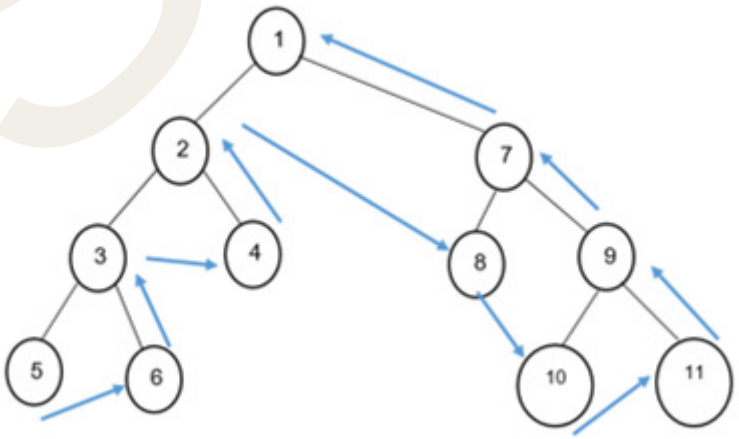


Fig 3.1.17 Postorder traversal

In the case of Figure 3.1.17, the Postorder traversal will give the following result.

L-R-V: Postorder traversal, that is, travel the left sub-tree (L), travel the right sub-tree (R) and process the visited node (V)

Algorithm: An algorithm for postorder travel (L-R-V) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```
PostOrderTravel(Tree){

if (Tree == NULL) return

else

{

postOrderTravel (leftChild (Tree));

postOrderTravel (rightChild (Tree));

process DATA (Tree);

}

}
```

Explanation: In the algorithm, an if else statement is included in the beginning to check if the tree is null. If the tree is NULL, we return. Else, the following steps are followed.

1. Traverse the left subtree, i.e., call postorderTravel (leftChild (Tree));
2. Traverse the right subtree, i.e., call postorderTravel (rightChild (Tree));
3. Visit the root i.e; process DATA (Tree);

Illustration :

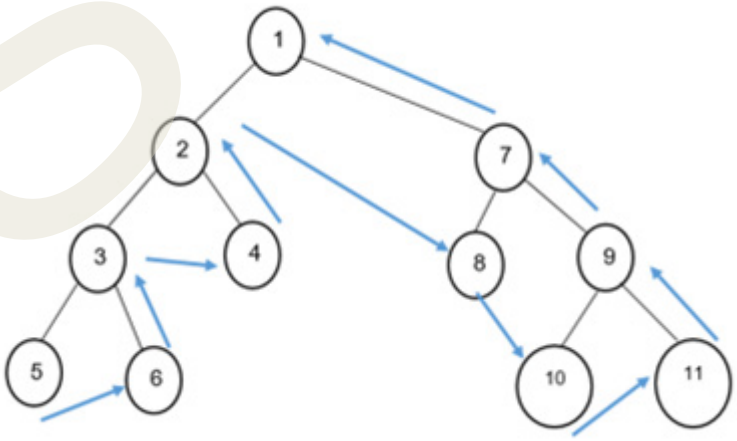


Fig 3.1.18 Postorder traversal

In the case of Figure 3.1.18, the Postorder traversal will give the following result.

5 6 3 4 2 8 10 11 9 7 1

Explanation: In the case of Postorder traversal, 5(L) is followed by 6(R), which is followed by 3(V). This is followed by 4(R), which is followed by 2(V). This is followed by 8(L), 10(L) and 11(R). This is followed by 9(V), 7(V), and 1(V).

Recap

- Trees are hierarchical non-linear data structures consisting of nodes connected by edges.
- Root - The node that has no parent. Each tree has a root node, which serves as the starting point.
- Leaf - The node that has no children.
- Degree - The number of children present in a node.
- Height - The number of nodes present in the longest path of the tree from root to a leaf node is called the height of the tree.
- Depth - The depth of a node is the length of its path from the root node.
- Siblings: Siblings are the nodes that share the same parent
- Complete binary tree – Here, all the nodes are filled except the last one, and the nodes must be left as much as possible at the last level.
- Full binary tree – In a full binary tree, every node other than the leaves has exactly two children, one on the left and one on the right, and each leaf node has no children.
- Perfect Binary Tree - In a perfect binary tree, all Internal nodes have two children, and all leaves are at the same level.
- Balanced Binary Tree - The height of the left and right subtrees of any node differ by utmost one.
- 2-tree or Extended binary trees – Here, each node has either 0 children or exactly 2 children.
- A tree node consists of:

  a. pointer that points towards the right node(Right Child Address)

  b. pointer that points towards the left node(Left Child Address)

c. data element.

- During tree traversal, all the nodes of a tree are visited, and their values may be printed.

Inorder traversal

1. travel the left sub-tree (L)
2. process the visited node (V)
3. travel the right subtree (R)

Preorder traversal

1. process the visited node (V)
2. travel the left sub-tree (L)
3. travel the right subtree (R)

Postorder traversal

1. travel the left sub-tree (L)
2. travel the right subtree (R)
3. process the visited node (V)

Objective Type Questions

1. What is the name of the node with no parents?
2. What is used to refer to the link from parent to child?
3. What is the name of the node without any children?
4. In which type of tree must the nodes be as left as possible in the last level.?
5. Which type of tree has the property that each node has either 0 children or 2 children?
6. What is the use of edges in a tree?
7. What is the other name for the leaf node?
8. What is the name given to the children of the same parent?

9. What is the second step in preorder traversal?

10. What is the first step in inorder traversal?

11. What is the third step in postorder traversal?

12. What is the maximum number of children that a binary tree can have?

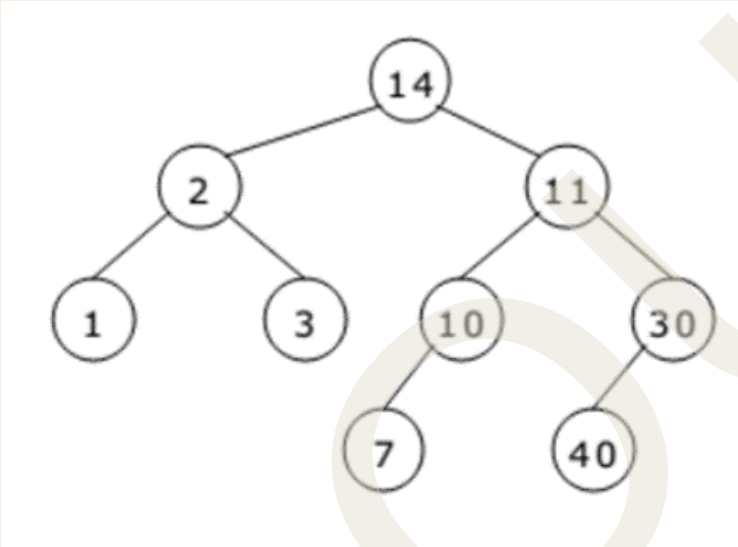


Fig 3.1.19 Question

13. How many leaf nodes are there for the above tree in Figure 3.1.19?

14. How many nodes have at least one sibling for the tree in Figure 3.1.19?

15. What is the depth of the tree in Figure 3.1.19?

16. What is the order of nodes visited using an Inorder traversal for the tree in Figure 3.1.19?

17. What is the order of nodes visited using a Preorder traversal for the tree in Figure 3.1.19?

18. What is the order of nodes visited using a Postorder traversal for the tree in Figure 3.1.19?

19. What is the minimum number of nodes with a full binary tree with a depth 3?

Answers to Objective Type Questions

1. Root node
2. Edge
3. Leaf
4. Complete binary tree
5. Extended binary tree
6. To connect nodes
7. Terminal node
8. Siblings
9. Traverse left subtree.
10. Traverse left subtree.
11. Visit the root.
12. Two
13. Four(4) [hint : 1,3,7,40]
14. Six(6) [hint : 2,11,1,3,10,30]
15. Three (3)
16. Inorder - 1 2 3 14 7 10 11 40 30
17. Preorder - 14 2 1 3 11 10 7 30 40
18. Postorder - 1 3 2 7 10 40 30 11 14
19. Fifteen (15)

Assignments

1. Identify the following from Figure 3.1.20

   i. Subtrees

ii. Number of levels in the tree

iii. Root of the tree

iv. Leaf nodes

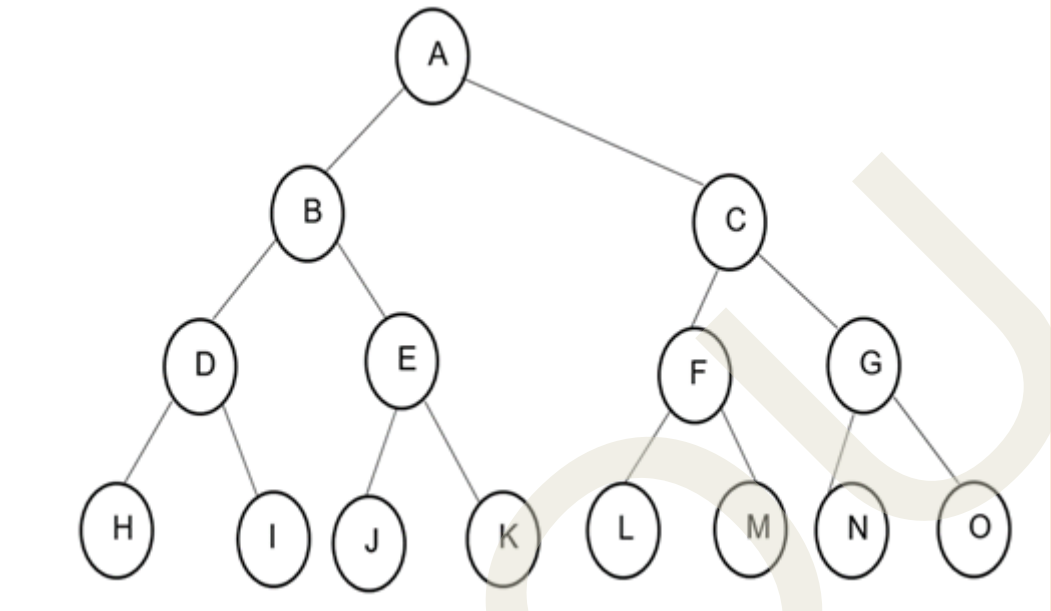


Fig 3.1.20 Question 1

2. Answer the following questions based on Figure 3.1.21

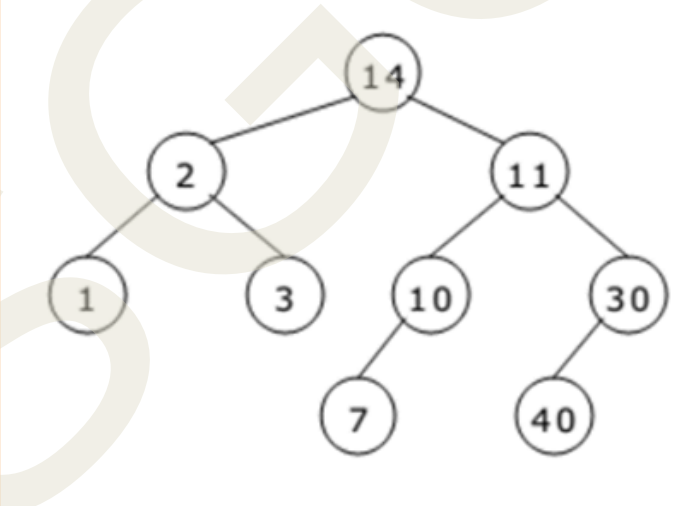


Fig 3.1.21

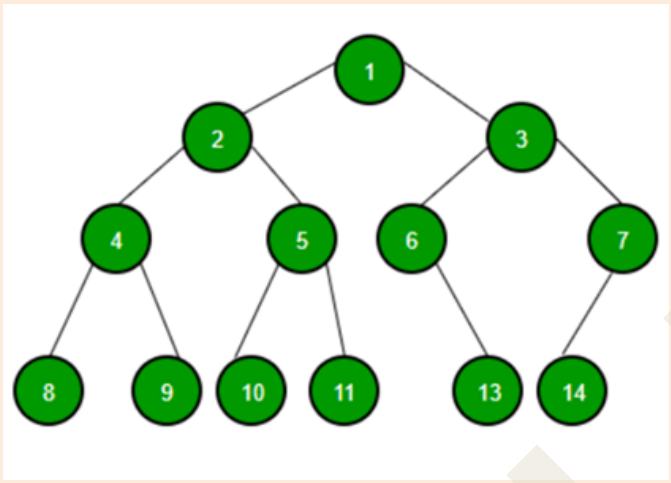
i. How many leaf nodes are there for the above tree in Figure 3.1.21?

ii. How many nodes have at least one sibling for the tree in Figure 3.1.21?

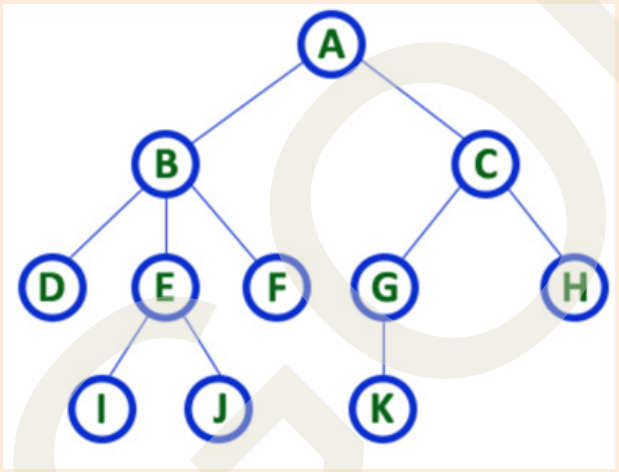
iii. How many descendants do not have the root for the tree in Figure 3.1.21?

iv. What is the depth of the tree in Figure 3.1.21?

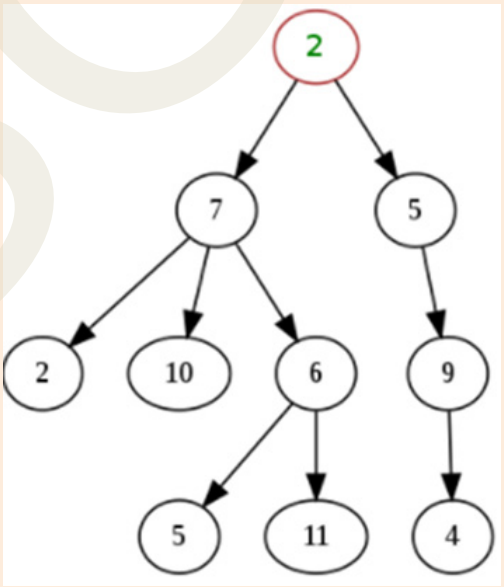
3. Identify the binary tree from the following figures



(i)



(ii)



(iii)

Fig 3.1.22 Question 3

4. What are the nodes to be removed from (i), (ii), or (iii) of Figure 3.1.22 to convert them to a binary tree?

5. Explain different tree traversals with examples.

6. Perform different tree traversal, inorder, preorder and postorder for the graph given below.

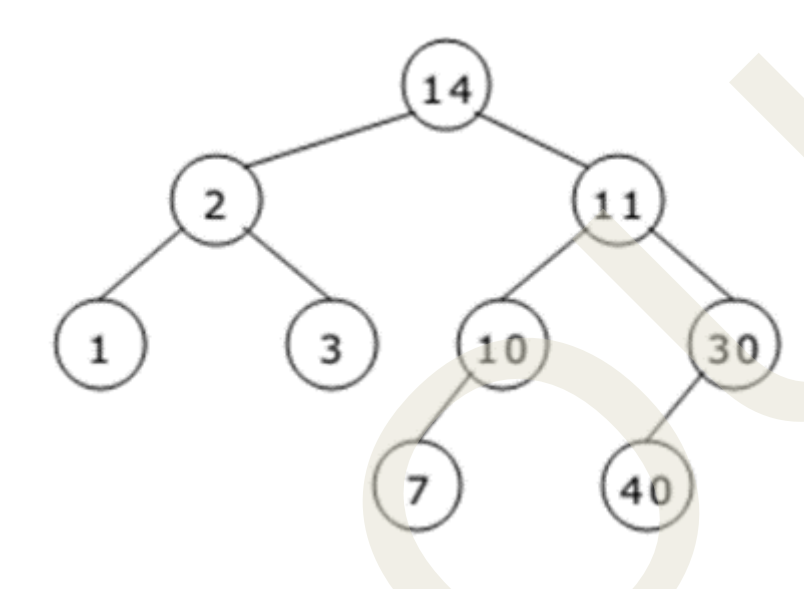


Fig 3.1.23

7. Explain different terminologies in binary trees with examples.

Suggested Reading

1. Sharma, A. K. Data Structures using C, 2e. Pearson Education India, 2013.

2. Kruse, Robert, and C. L. Tondo. Data structures and program design in C. Pearson Education India, 2007.

3. Mark, Allen Weiss. "Data structures and algorithm analysis in C." (1992).

4. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. Data structures and algorithms. Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.

UNIT 2 Binary Search Tree

Learning Outcomes

Upon completion of this Unit, the learner will be able to:

- familiarise with the concept of the Binary Search Tree and its properties.
- make aware of searching in binary search trees.
- study the methods for insertion operation in binary search trees.
- learn the steps for deletion operation in binary search tree

Prerequisites

Let us again take the example where you were given the task of doing system maintenance at three offices in different destinations, namely Kollam, Kozhikode, and Thiruvananthapuram. Now, instead of writing the places, if we substitute them with the district IDs (keys), Cochin can be replaced by 7, Thiruvananthapuram by 1, and Kozhikode by 11. Let us represent these district IDs in the form of a binary search tree. In the case of a binary search tree, the left subtree key value will be less than root and the right subtree key value will be greater than root.

Figure 3.2.1 is an example of a binary search tree representation of three districts with their district IDs.

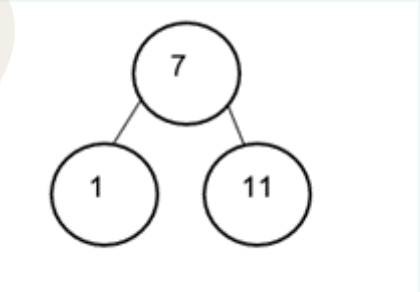


Fig 3.2.1 Binary Search Tree

The inorder traversal with district IDs will give the following result: 1 7 11.

Inorder traversal of a tree will result in the numbers being in ascending order."

Keywords

Binary search tree operations, Search algorithm, Insertion algorithm and Deletion algorithm.

Discussion

3.2.1 Introduction to Binary Search Tree

The following Figure 3.2.2 shows the districts of Kerala state and a binary search tree for the entire state of Kerala that is created using district IDs.

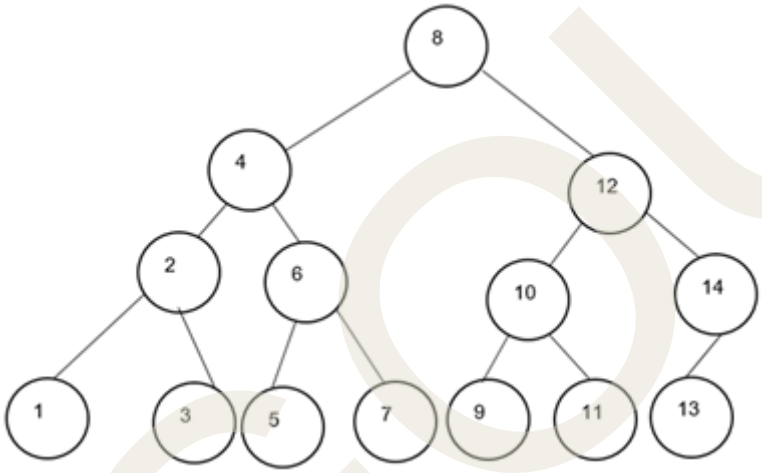


Fig 3.2.2 Binary Search Tree created using District IDs.

In Figure 3.2.2, we find that the root node (8) is greater than all the left child nodes and that the right subtree of the root node has a value greater than 8. This is essentially the property of a binary search tree.

The inorder traversal with all district IDs will give the following result, which is in sorted order:

1 2 3 4 5 6 7 8 9 10 11 12 13 14

However, the binary search tree is a little tricky when it comes to the insertion or deletion of a number or while searching for a number.

Let us consider any of the following hypothetical situations where

1. District 2 is removed from the figure 3.2.2
2. District with id 0 is added to the figure 3.2.2
3. Search for the district with id 9 in the figure 3.2.2

There are specific algorithms that can be used to insert, delete, or search for a number in a binary search tree. We will be discussing them in the following sections. In summary,

a binary search tree (BST) is a special binary tree. It is used for efficient data searching.

3.2.1.1 Properties of Binary Search Tree

Let us consider our binary search tree in the figure 3.2.2. Here, we find that the root node(8) is greater than all the left child nodes and that the right subtree of the root node has a value greater than 8.

Thus, in binary search trees, all the left subtree elements should be less than the root data, and all the right subtree elements should be greater than the root data. This is called the binary search tree property. However, this property should be satisfied at every node in the tree.

- The data to be stored must have unique key values.
- The left subtree of a node contains only nodes with keys lesser than the root node's key.
- The right subtree of a node contains only nodes with keys greater than the root node's key.
- Each left and right subtree must also be a binary search tree.

In simple words, we can say that the key contained in the left child is less than the key of its parent node (here 108), and the key contained in the right child is more than the key of its parent node, as shown in Figure 3.2.3.

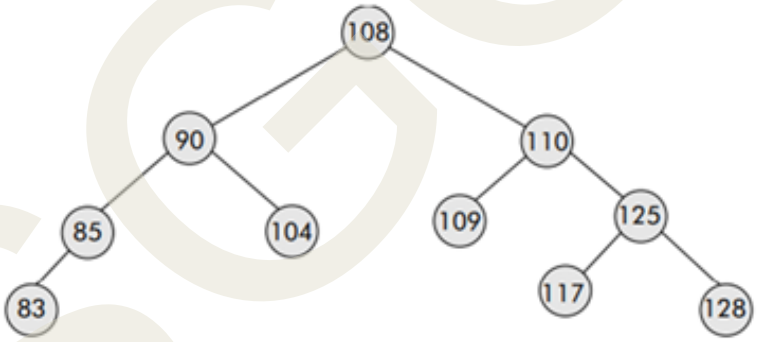


Fig 3.2.3 Binary Search Tree

The inorder travel of the binary search tree would produce the following node sequence:

83 85 90 104 108 109 110 117 125 128

The inorder list thus generated will produce a sorted list. Since a binary search algorithm can be easily applied here, it is called a binary search tree.

3.2.2 Searching in a Binary Search Tree

The most important application of a BST is searching for a value. In the case of binary search trees, searching is faster compared to binary trees because everything is stored in sorted order.

In a BST, the data becomes sorted during its creation. The item to be searched is compared with the root of the BST. If it matches the root, success is returned. Otherwise, the left or right subtree is searched depending on whether the item is less than or greater than the root.

Algorithm: An algorithm for searching a BST is given below; it searches for a value in a BST pointed to by a pointer called 'Tree'. This is a recursive algorithm.

```
searchBST(Tree, Val){
if (Tree = NULL)
return failure;
if (DATA (Tree) == val)
return success;
else
if (Val < DATA (Tree))
searchBST(Tree-> leftChild, Val);
else
searchBST(Tree-> rightChild, Val);
}
```

Explanation: In the algorithm, an if statement is included in the beginning to check if the tree is null. If the tree is NULL, the algorithm returns a failure. Else, the following steps are followed.

1. The item to be searched(val) is compared with the root of the BST, i.e, (DATA (Tree) == val). If the condition is true, the algorithm returns success.
2. If the item to be searched(val) is less than the root, search the left sub tree. i.e.,

   ```
   if (Val < DATA (Tree))
   BSTsearch (Tree->leftChild, Val);
   ```

3. Else search the right subtree. i.e.,

   ```
   else BSTsearch (Tree->rightChild, Val);
   ```

Illustration: The following Figure 3.2.4 shows how to search for a number.

Suppose we want to search for 76. The algorithm will first check if the tree is null. Since the tree is not null, the item to be searched (here76) is compared with the root of the BST, i.e, (here 57). Since the condition is false, the algorithm checks if the item is less than the root; since 76 is greater than 57, the algorithm searches the right subtree till the

number is obtained.

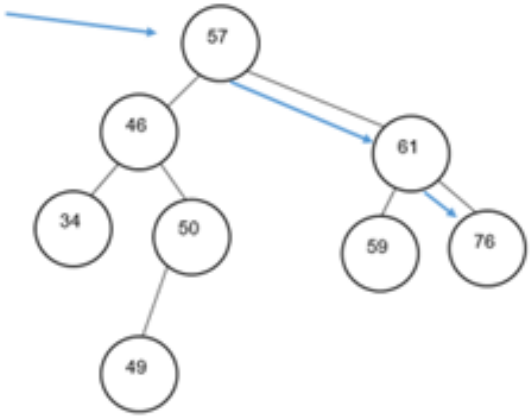


Fig 3.2.4 Searching a BST

3.2.3 Insertion operation in BST

The insertion operation in BST is basically a search operation. The idea is if the item to be inserted is already found in the BST, the insertion operation fails. On the other hand, if a null pointer is encountered, details of which will be discussed while discussing the algorithm, the item is attached there itself.

Algorithm: An algorithm for insertion of an item into a BST is given below:

```
insertBST (binTree, item){
if (binTree == NULL) {
Take node in ptr;   /* Define a pointer ptr for the node  */
DATA (ptr) = item;
leftChild(ptr) = NULL;
rightChild(ptr)= NULL;
binTree = ptr;
return success;
}
else
if (DATA (binTree) == item)
return failure;
else {
Take node in ptr;      /* Define a pointer ptr for the node  */
DATA (ptr) = item;
```

leftChild (ptr) = NULL;

rightChild(ptr)= NULL;

if (DATA (binTree) < item)

insertBST (binTree->leftChild, ptr);

else

insertBST (binTree->rightChild, ptr);

}

}

Explanation: In the algorithm, an if statement is included in the beginning to check if the tree is null. If the tree is NULL, the algorithm creates a new binary search tree with a pointer for leftChild and rightChild and returns success. i.e

Take node in ptr;

DATA (ptr) = item;

leftChild(ptr) = NULL;

rightChild(ptr)= NULL;

binTree = ptr;

return success;

If the item to be inserted is already found in the BST, the insertion operation fails. i.e.,

else

if (DATA (binTree) == item)

return failure;

Else, if a null pointer is encountered, the item is attached there itself. The left child pointer and right child pointer are made null, i.e.,

else

Take node in ptr;

DATA (ptr) = item;

leftChild (ptr) = NULL;

rightChild(ptr)= NULL;

Else, if the item to be inserted is less than the root, insert the item as the left child, i.e.,

if (DATA (binTree) < item)

insertBST (binTree->leftChild, ptr);

Else, insert the item as the right child, i.e.,

insertBST (binTree->rightChild, ptr);

Illustrations:

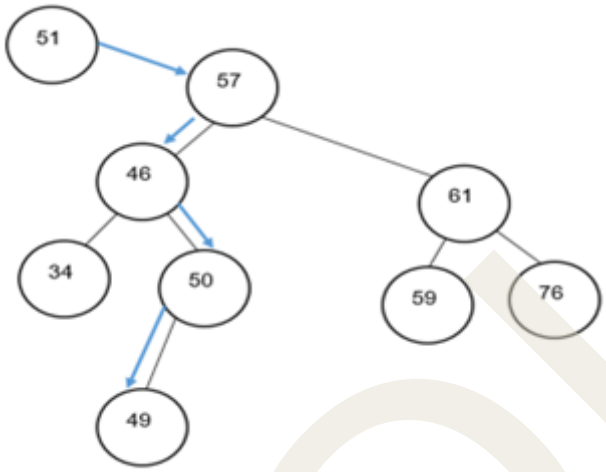


Fig 3.2.5 Process of insertion of number 51

In the above Figure 3.2.5, number 51 is inserted into the BST. It is searched in the BST, and it comes to a dead end at the node containing 49.

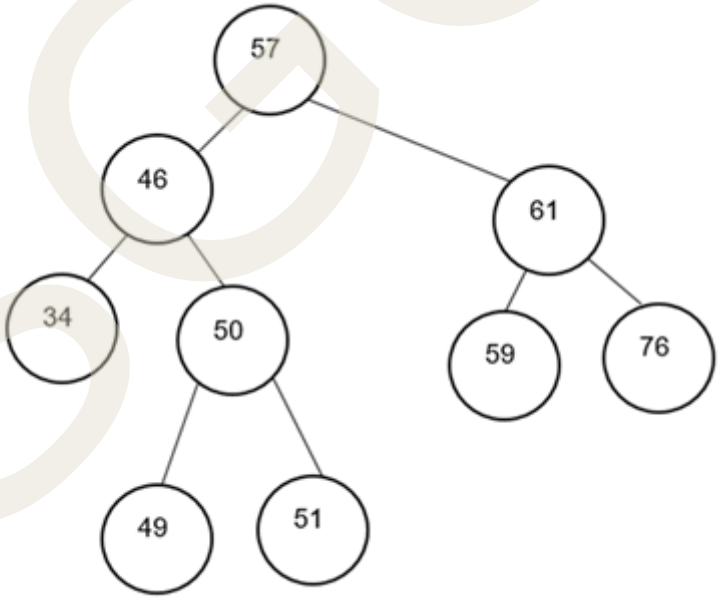


Fig 3.2.6 Insertion operation in BST

In the above figure 3.2.6, since the right child of 50 is NULL, the number 51 is attached there itself.

3.2.4 Deletion operation in BST

In the case of deletion of a node in BST, 3 cases are possible:

Case 1: The node is a leaf node.

Case 2: The node has only one child.

Case 3: The node is an internal node, i.e., having both the children

Case 1: The node is a leaf node.

In this case, the pointer to the node from its parent is set to Null, and then the node is freed.

This is illustrated in the following Figure 3.2.7.

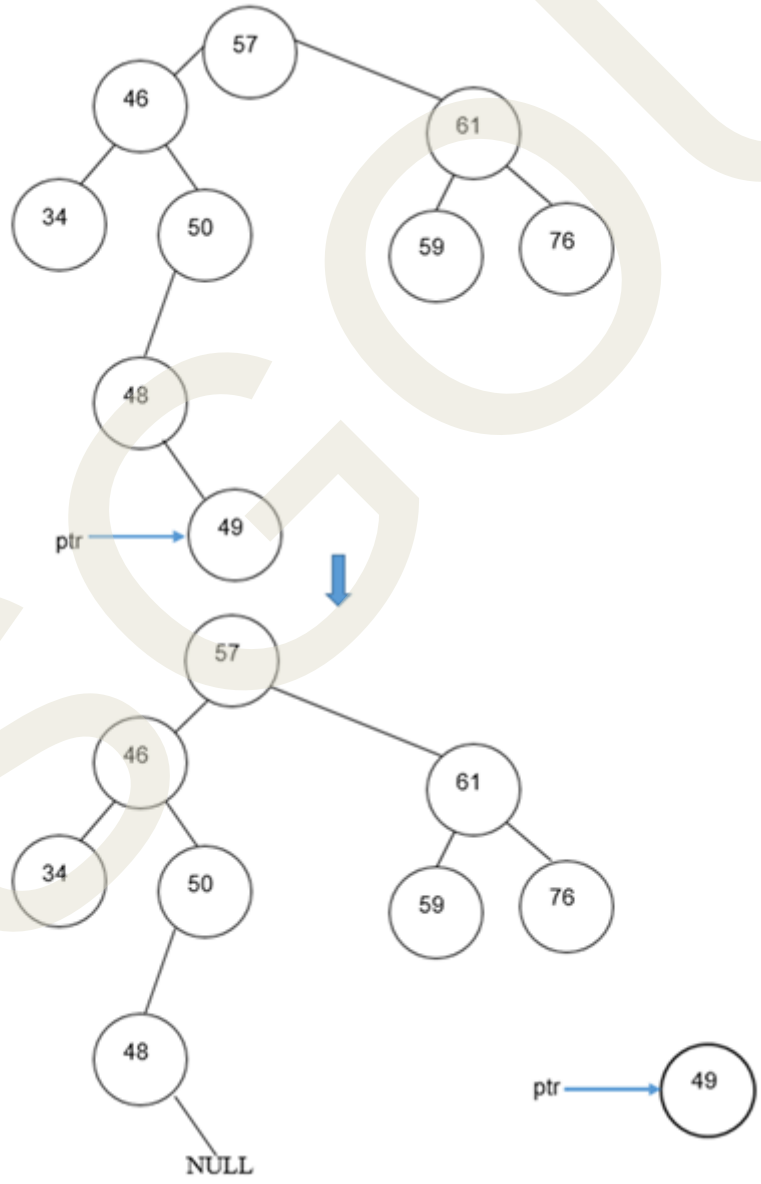


Fig 3.2.7 Deletion of a leaf node

This case can be very easily handled by setting the pointer to the node from its parent equal to NULL and freeing the node, as shown in Figure 3.2.7. Here, the leaf node containing 49 has been deleted, its parent's pointer has been set to NULL, and the node itself has been freed.

Case 2: The node has only one child

This case can also be handled very easily by setting the pointer to the node from its parent equal to its child node. The node may be freed later on, as shown in Figure 3.2.8. The node containing 48, having only one child (i.e., 49), has to be deleted. In order to accomplish this, the parent's pointer (i.e., 50) of node 48 has been set to node 48's child (i.e., 49), and the node itself has been freed.

This has been illustrated in the following Figure 3.2.8

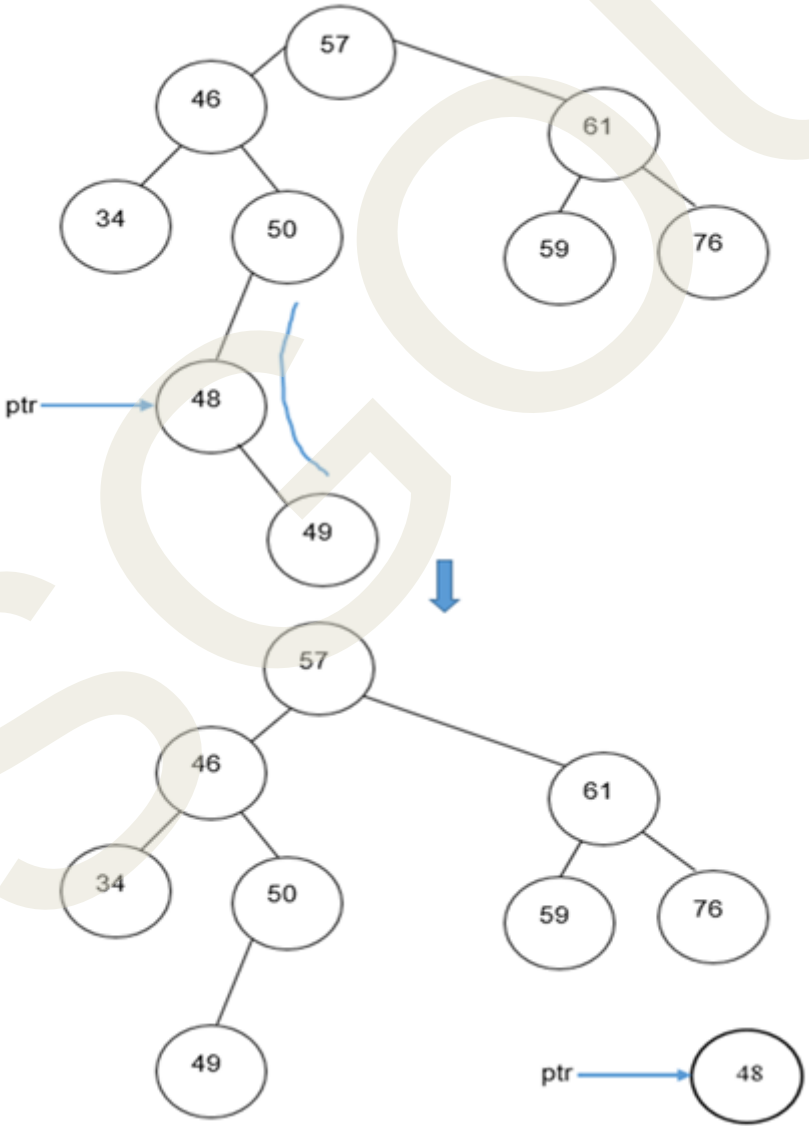


Fig 3.2.8 Deletion of a node having only one child

Case 3: The node is an internal node, i.e., having both the children

If the node to be deleted is an internal node (i.e., having both children), find the inorder successor of the node. Copy the contents of the inorder successor to the node to be deleted and delete the inorder successor.

Algorithm: An algorithm to find the inorder successor of a node is given below.

```
findSucc(ptr) {
 succPtr = rightChild (ptr);
 while (leftChild (succPtr != NULL))
 succPtr = leftChild (succPtr);
 return succPtr;
}
```

Explanation: The algorithm uses a pointer called succPtr that travels from ptr to its successor and returns it. Following figure 3.2.9 shows the illustration.

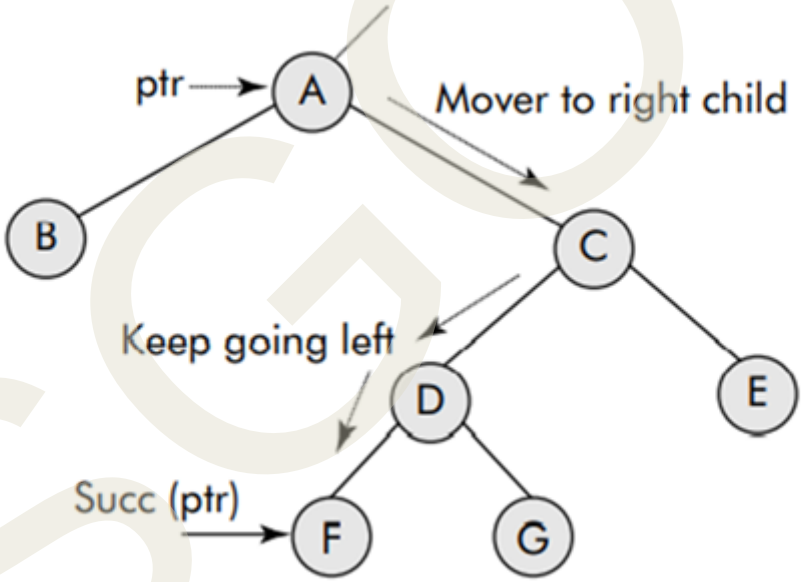


Fig 3.2.9 Finding inorder successor of a node

The successor of a node can be reached by first moving to its right child and then continuing to left till a NULL is found, as shown in Figure 3.2.9. The successor node is always present in the right subtree of the internal node. Note that the successor node will be the least value in the right subtree of the internal node; if an internal node ptr is to be deleted, then the contents of inorder successor of ptr should replace the contents of ptr and the successor be deleted by the methods suggested for Case 1 or Case 2. (If a node has a right child, the successor is the minimum value node in the right subtree of that node). The mechanism is shown in Figure 3.2.10. The explanation of the mechanism is given in the next page.

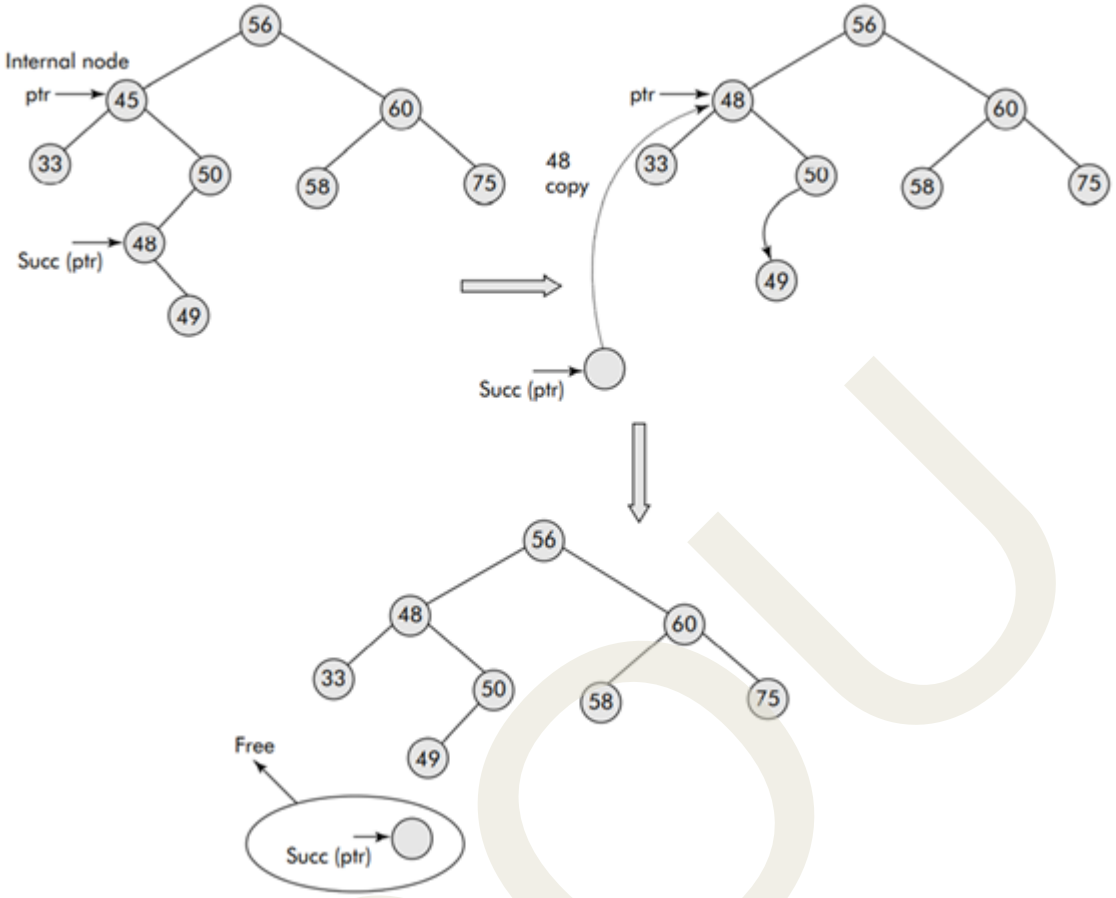


Fig 3.2.10 Deletion of an internal node in BST

```
del NodeBST(Tree, Val){
ptr = search (Tree, Val); /* Finds the node to be deleted */
parent = search (Tree, Val); /* Finds the parent of the node */
if (ptr == NULL) report error and return
if (leftChild (ptr) == NULL &&; rightChild(ptr) == NULL) /* Case 1*/
{
if (leftChild (Parent) == ptr)
leftChild (Parent ) = NULL;
else
if (rightChild (Parent) == ptr)
rightChild (Parent ) = NULL;
free (ptr);
}
```

```
if (leftChild (ptr ) != NULL &&; rightChild(ptr) == NULL) /*Case 2 */
{
if (leftChild (Parent) == ptr)
leftChild (Parent ) = leftChild (ptr);
else
if (rightChild (Parent) == ptr)
rightChild (Parent ) = leftChild (ptr);
free (ptr);
}
else
if (leftChild (ptr ) == NULL &&; rightChild(ptr) != NULL) /*Case 2 */
{
if (leftChild (Parent) == ptr)
leftChild (Parent ) = rightChild (ptr);
else
if (rightChild (Parent) == ptr)
rightChild (Parent ) = rightChild (ptr);
free (ptr);
}
if (leftChild (ptr ) != NULL &&; rightChild(ptr) != NULL) /*Case 3 */
{
succPtr = findSucc (ptr);
DATA (ptr) = DATA (succPtr);
delNodeBST (Tree, succPtr);
}
```

Explanation:

Case 1: The pointer to the node from its parent is set to Null, and then the node is freed. i.e., If the node is a leaf node, delete the node directly.

Case 2: The pointer to the node from its parent is set equal to its child node. i.e., Else, if the node has one child, copy the child to the node to be deleted and delete the child

node.

i.e. if the node to be deleted has one child, copy the contents of the child to the parent and then delete the child.

Case 3: If the node to be deleted is an internal node (i.e., having both children), find the inorder successor of the node. Copy the contents of the inorder successor to the node to be deleted and delete the inorder successor.

Recap

- In binary search trees, all the left subtree elements should be less than the root data, and all the right subtree elements should be greater than the root data.
- The data to be stored in a BST must have unique key values.
- The data automatically becomes sorted during the creation of a Binary Search Tree(BST).
- The most important application of BST is for searching for a value.
- The insertion operation in BST is basically a search operation.
- In the case of deletion of a node in BST, 3 cases are possible:

  Case 1: The node is a leaf node.

  Case 2: The node has only one child.

  Case 3: The node is an internal node, i.e., having both the children

Objective Type Questions

1. Name one of the most efficient tree data structures.
2. Which traversal technique lists the nodes of a binary search tree in ascending order?
3. What is the main advantage of searching in a BST compared to a binary tree?
4. What is the key property of BST?
5. Which node in a binary search tree contains the smallest key?
6. Which node in a binary search tree contains the highest key?
7. In a BST, which of the following operations does not require rebalancing?
8. In the case of node deletion in a BST, how many possible cases are there.

9. What type of node has both children?

10. What is in-order successor of a BST?

Answers to Objective Type Questions

1. BST (Binary Search Tree)

2. Inorder.

3. Sorted

4. The left subtree of a node contains only nodes with keys less than the node keys, and the right subtree of a node contains only nodes with keys greater than the node keys

5. Left most node

6. Rightmost node

7. Search

8. Three

9. Internal

10. Left most node in right sub tree

Assignments

1. Explain the algorithm for the insertion of a node into a binary search tree. Give illustration.

2. Explain the algorithm for the deletion of a node from a binary search tree. Give illustration.

3. Explain the algorithm for searching a node into a binary search tree. Give illustration

4. Explain the features of BST with examples.

5. Perform inorder, preorder and postorder traversal of BST.

6. What are the benefits of searching, inserting, and deleting in a BST?

Suggested Reading

1. Sharma, A. K. Data Structures using C, 2e. Pearson Education India, 2013.
2. Kruse, Robert, and C. L. Tondo. Data structures and program design in C. Pearson Education India, 2007.
3. Mark, Allen Weiss. "Data structures and algorithm analysis in C." (1992).
4. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. Data structures and algorithms. Vol. 175. Boston, MA, USA:: Addison-Wesley, 1983

UNIT 3

Balancing Binary Tree

Learning Outcomes

Upon completion of this Unit, the learner will be able to:

- study the concept of height-balanced trees
- make aware of balanced binary search trees
- gain awareness of AVL trees and its important operations
- unroll the concept of B-tree

Prerequisites

Let us consider a township where A is the capital city. A has two neighbouring cities, B and C, as shown in Figure 3.3.1. D and E are the neighbouring cities of B, while C has F as a neighbouring city. The neighboring cities of D are G and H, while the neighboring city of G is I.

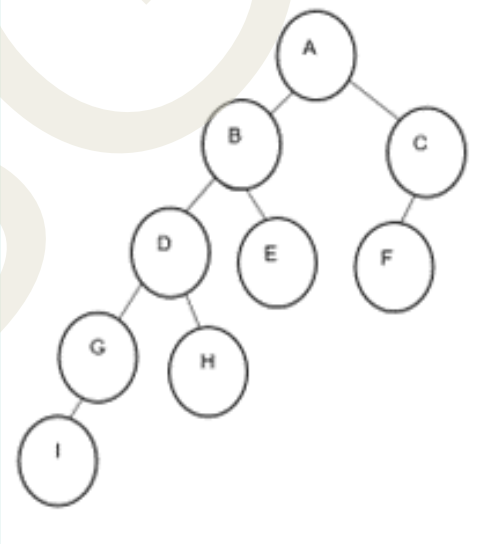


Fig 3.3.1 Mobile route tree

Let us now consider an example where the messages are passed between the cities from people living in city A to people living in other cities. The following Figure 3.3.2 shows

the hypothetical towers that are assigned to each city. From Figure 3.3.2, the allocation of towers is as follows:

City A - Tower 1

City B and C - Tower 2

City D, E, F - Tower 3

City G, H - Tower 4

City I - Tower 5

If we consider the cities as nodes, figure 3.3.1 can be represented in the following manner.

The following figure 3.3.2 shows the cities as nodes, the depth of the tree, and the allocated towers. The following figure 3.3.2 represents the depth of the tree and the Tower Levels.

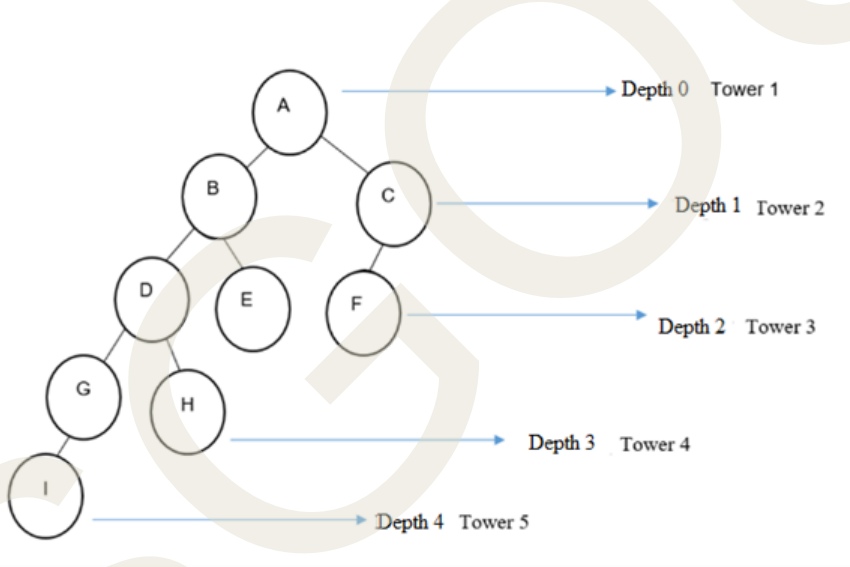


Fig 3.3.2 Mobile tree tower with subnodes

Suppose a message needs to be sent from Tower A to node G and to node F. In order for the information to reach node I, the message has to be passed through several towers while with the case of node F, information reaches F by passing through one tower in between. This is a possible imbalance as there will be a delay in the reach of message between the people living in the same township, though under different cities. Inorder to avoid such circumstances, tree balancing is performed.

Key Concepts

Balanced binary tree, AVL tree, Insertion algorithm, Balancing procedure, B-tree

Discussion

3.3.1 Concept of Balancing Tree Data Structure

Figure 3.3.1 is an example of an unbalanced tree. Due to imbalances in the tree, there will be a delay in the transmission of information. In order to avoid such circumstances, the concept of balancing is introduced.

A balanced tree is highly efficient when operations like insertion, deletion, etc., compared to unbalanced trees. In order for a tree to be perfectly height balanced, its left subtree and right subtree should be at the same level. The following Figure 3.3.3 shows a perfectly balanced tree.

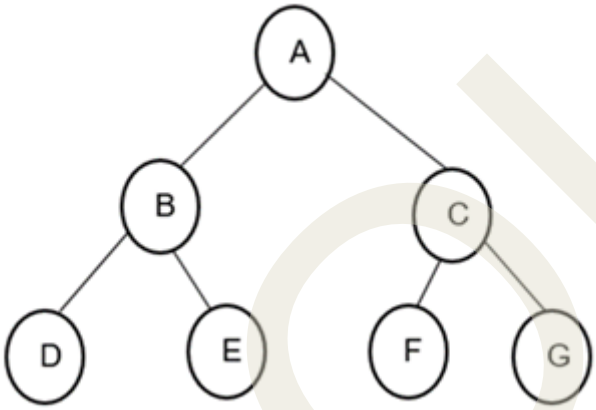


Fig 3.3.3 Perfectly balanced tree

In the above figure 3.3.3, the left subtree with root B and the right subtree with root C are at the same height. Hence, figure 3.3.3 is a perfectly balanced tree. However, perfectly balanced trees are rare. An alternative is 'almost' a perfectly balanced tree. A tree is said to be height-balanced if the heights of the left and right subtrees of each node are within 1. The following figure shows an example of a height-balanced tree that fits the definition of an 'almost' perfectly balanced tree.

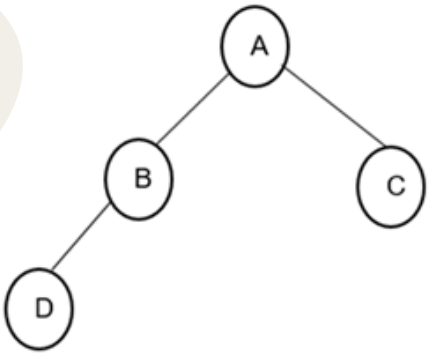


Fig 3.3.4 Height-balanced tree

In the above figure 3.3.4, the difference between the left sub-tree and the right subtree of each node is within the range 1. So, the tree is called a height-balanced tree.

3.3.1.1 Advantages of Balancing Tree Data Structure

The aim of a balanced tree is to reach the leaf in a minimum of traversal (min height). Thus, it helps in better searching.

Another advantage of balanced binary search trees is their ability to maintain a dynamic set in sorted order and, at the same time, support a large range of operations.

3.3.2 Balanced Binary Tree

The search time in a BST depends upon its height. If the heights of both the left and right subtrees of any node are equal, then the searching is efficient. However, when we insert and delete contents from the BST, the BST is likely to become unbalanced. The efficiency of searching is considered to be ideal if the difference between the heights of the left and right subtrees of all the nodes in a BST is at most one. Such a binary search tree is called a Balanced BST.

The following are the conditions for a height-balanced binary tree:

- The difference between the left and the right subtree for any node is not more than one
- The left subtree is balanced
- The right subtree is balanced

3.3.3 AVL Tree

When the kids are playing with building blocks, when they want to make a models, as shown below, the chances of falling is more in figure 3.3.5 when compared that of figure 3.3.6.

In Figure 3.3.5, the difference between the heights of blocks on the left side and the right side is 2, while in Figure 3.3.6, the difference between the heights of the blocks on the left side and the right side is 1. Hence, figure 3.3.6 is more stable than figure 3.3.5.

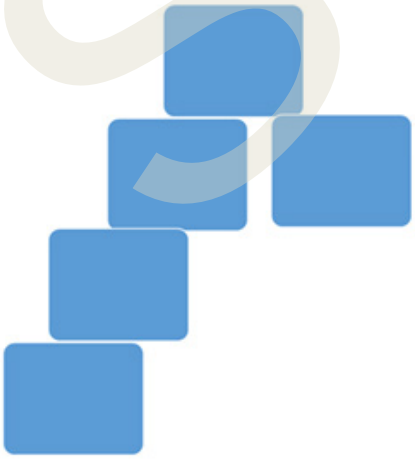


Fig 3.3.5 unbalanced building blocks

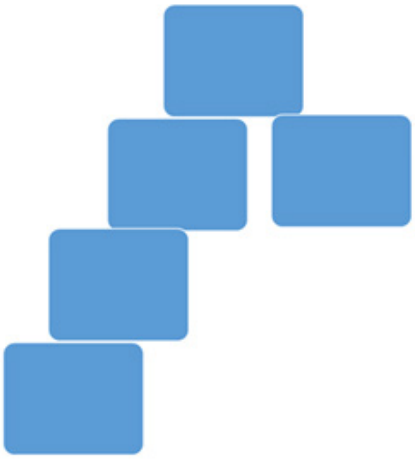


Fig 3.3.6 Balanced building block

The above scenario can be compared with AVL trees. AVL tree got its name after its inventor, Georgy Adelson-Velsky and Landis. An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1. It is thus a balanced binary search tree.

In summary, the AVL tree is a self-balancing binary search tree. In working with AVL trees, operations must preserve two properties:

- (height-balanced) heights of left and right subtrees are within 1
- (BST) values in the left subtree are smaller than the root value, and values in the right subtree greater than the root

Consider the binary trees shown in Figures 3.3.7 and 3.3.8

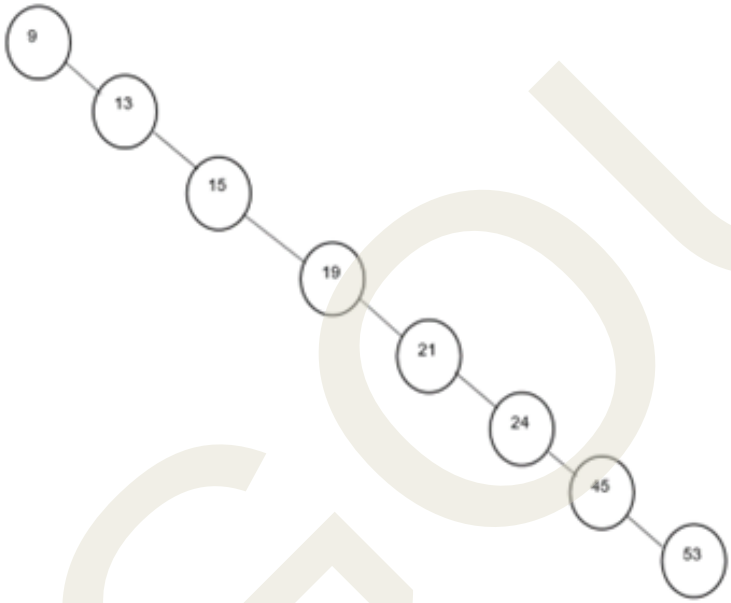


Fig 3.3.7 Tree Example 1

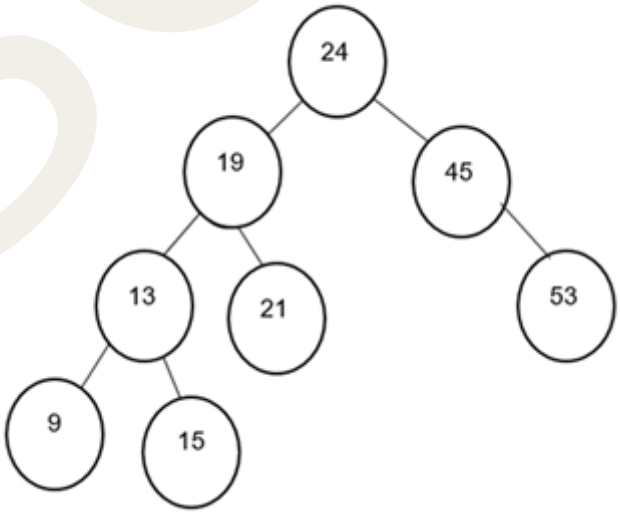


Fig 3.3.8 Tree example 2 (AVL tree)

In the case of Figure 3.3.7, it requires

2 tests to locate 13

3 tests to locate 15

8 tests to locate 53

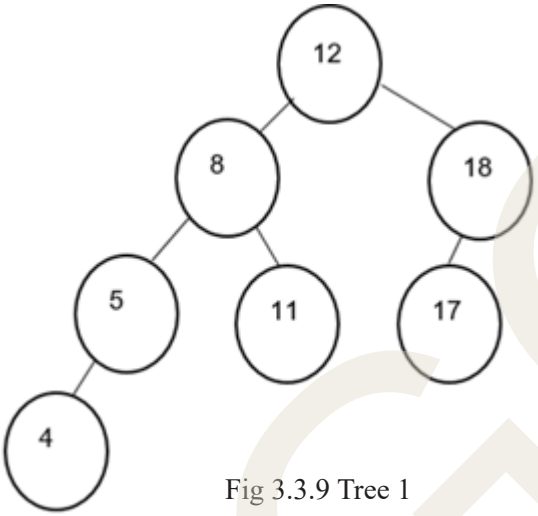
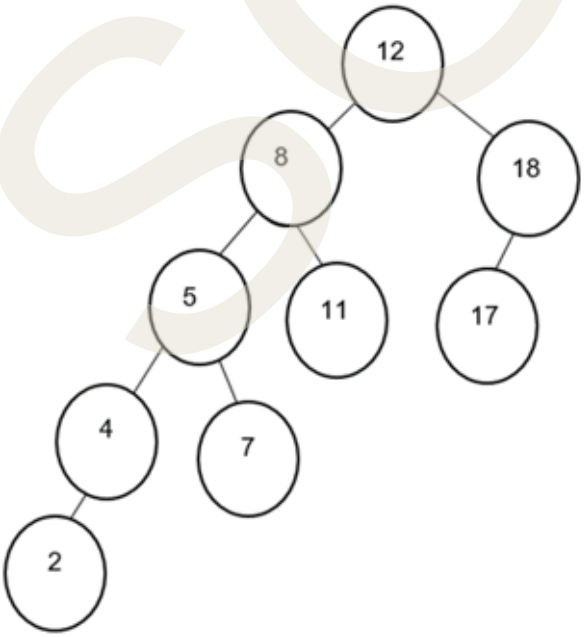
In the case of Figure 3.3.8, it requires

4 tests to locate 9 and 15

3 tests to locate 21 and 53

i.e., the maximum search effort for the tree in Figure 3.3.8 is either three or four. We thus see that balancing a tree can lead to significant performance improvements.

Now, let us consider another tree example to understand the AVL tree in detail.

Tree	Explanation
 Fig 3.3.9 Tree 1	Since each left subtree has a height greater than each right subtree, Tree 1 is an AVL tree.
 Fig 3.3.10 Tree 2	Since the sub-tree with node 8 has a height of 4 and the sub-tree with node 18 has a height 2, Tree 2 is not an AVL tree.

One of the applications of AVL trees is in telephone line connections. We have discussed this in detail in the above section.

3.3.3.1 Balancing Factor

We usually denote an AVL tree along with its balancing factor. Balancing Factor (BF) is defined as given below:

BF = height of left subtree (H_L) - height of right subtree (H_R)

The following Figure 3.3.11 shows an AVL tree with BF.

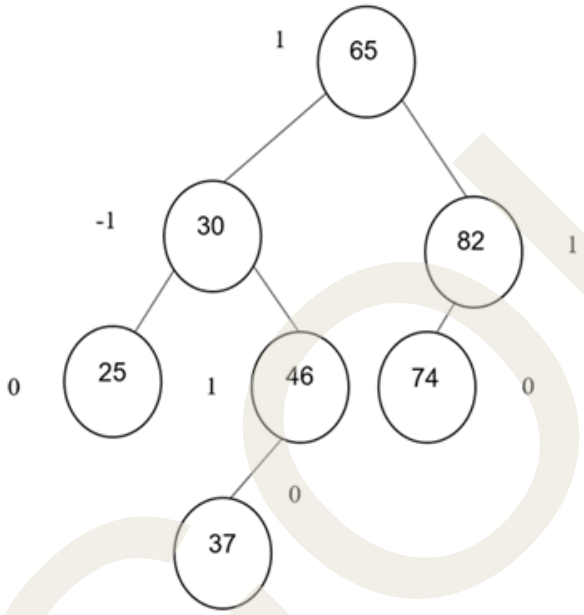


Fig 3.3.11 AVL tree with BF

In the above figure 3.3.11, node 37 is a leaf node and hence is given BF of 0. In the case of the left sub tree with root 30, node 25 is a leaf node and hence given BF of 0. Node 46 has a left child but does not have a right subtree. Therefore BF =1. The BF of node 30 is 0 – 1 = -1. In the same manner, the leaf node 74 has a BF of 0. Node 82 has a left child but does not have a right child. Hence BF=1. The BF of the root node 65 = height of left subtree (H_L) - the height of right subtree (H_R) = 2 – 1 = 1.

3.3.3.2 Unbalanced Trees

As we discussed earlier, the tree becomes unbalanced whenever a node is inserted into a tree or whenever a node is deleted from a tree. When it is detected that the tree is unbalanced, we need to rebalance it. In the case of AVL trees, balancing is performed by rotating nodes either to the left or to the right. Following are categories of unbalanced trees:

Case 1: Left of left (LL) Unbalanced

Case 2: Right of right (RR) Unbalanced

Case 3: Right of left (RL) Unbalanced

Case 4: Left of right (LR) Unbalanced

Case 1: Left of Left

Before

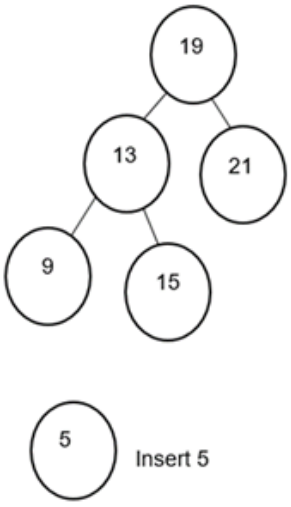


Figure 3.3.12(a)

After

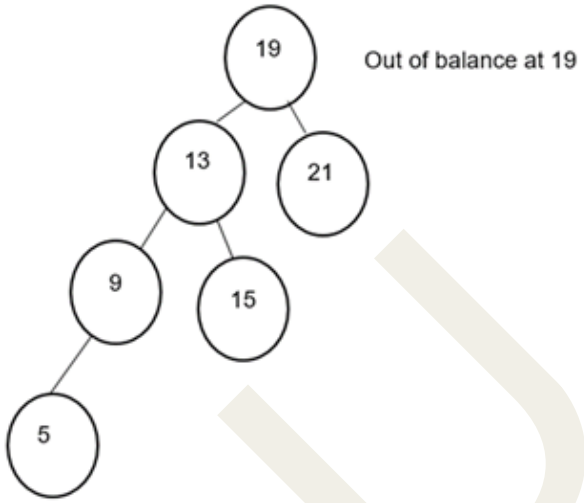


Figure 3.3.12(b)

1. Figure 3.3.12(a) is a balanced tree as the difference between the height of the left subtree and the right subtree is 1. In the above figure 3.3.12(a), node 5 is to be inserted below node 9, resulting in figure 3.3.12(b). Figure 3.3.12(b) is out of balance at 19 as the difference between the left sub tree and the right subtree is more than 1. Initially,
2. Figure 3.3.12(a) is Left high. It is again left high after the insertion of 9. Hence, the name Left of Left.

Case 2: Right of Right

Before

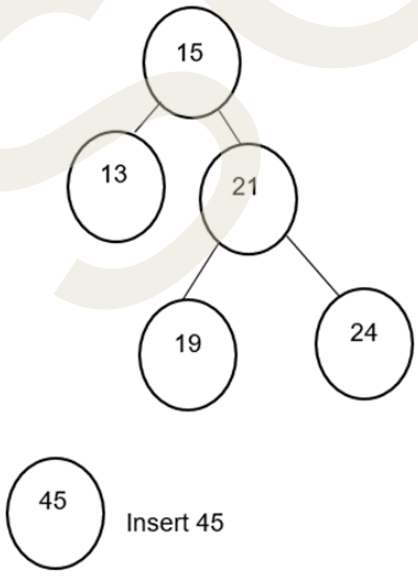


Fig 3.3.13(a)

After

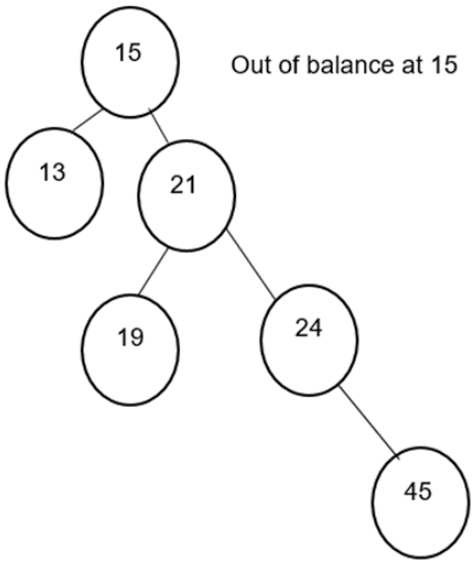


Fig 3.3.13(b)

In the figure 3.3.13(a), node 45 is to be inserted below node 24, resulting in figure 3.3.13(b). Figure 3.3.13(b) is out of balance at 15 as the difference between the left sub tree and the right subtree is more than 1. Initially, Figure 3.3.13(a) is Right high. It is again Right high after the insertion of 45. Hence the name Right of Right.

Case 3: Right of Left

Before

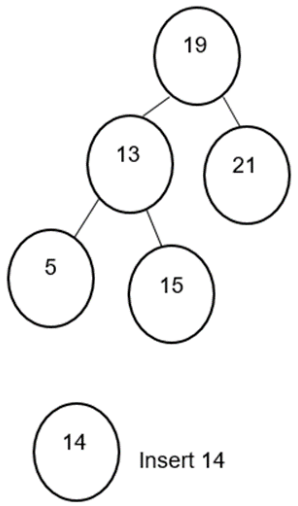


Fig 3.3.14(a)

After

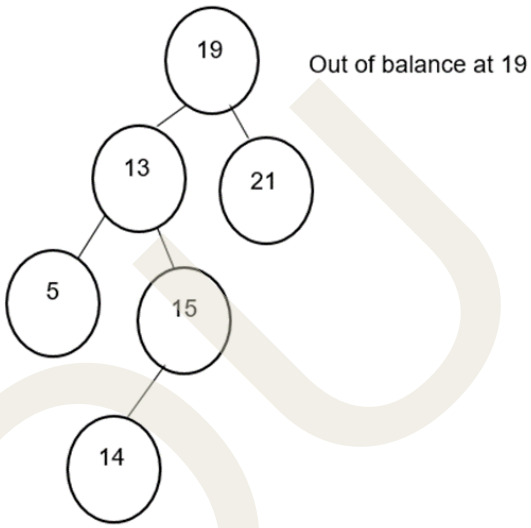


Figure 3.3.14(b)

In the above figure 3.3.14(a), node 14 is to be inserted below node 15 as the Left child, resulting in figure 3.3.14(b). Figure 3.3.14(b) is out of balance at 19 as the difference between the left sub tree and the right subtree is more than 1. Initially, Figure 3.3.14(a) is left high. Here, a subtree that was left high has become right of left due to the insertion of 14. Hence, the name Right of Left.

Case 4: Left of Right

Before

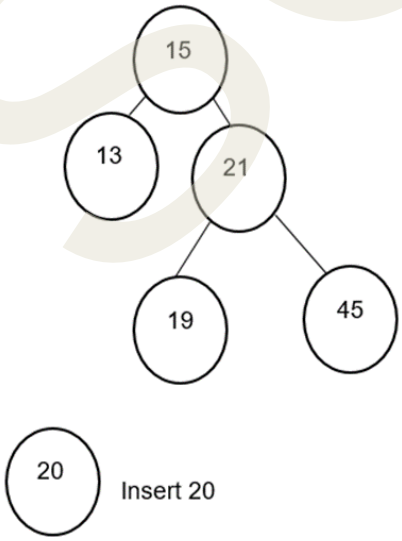


Figure 3.3.15(a)

After

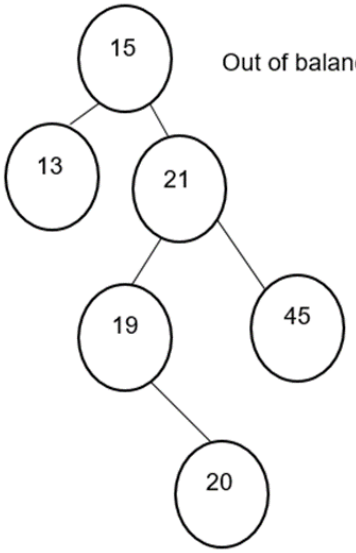


Figure 3.3.15(b)

In the figure 3.3.15(a), node 20 is to be inserted below node 19 as the Right child, resulting in figure 3.3.15(b). Figure 3.3.15(b) is out of balance at 15 as the difference between the left sub tree and the right subtree is more than 1. Initially, Figure 3.3.15(a) is right high. Here, a subtree that was right high has become left of right high due to the insertion of 20. Hence the name Left of Right.

Let us now discuss the ways to balance the out-of-balance trees with respect to case 1, case 2, case 3, and case 4.

Case 1: Left of Left Balancing

Before

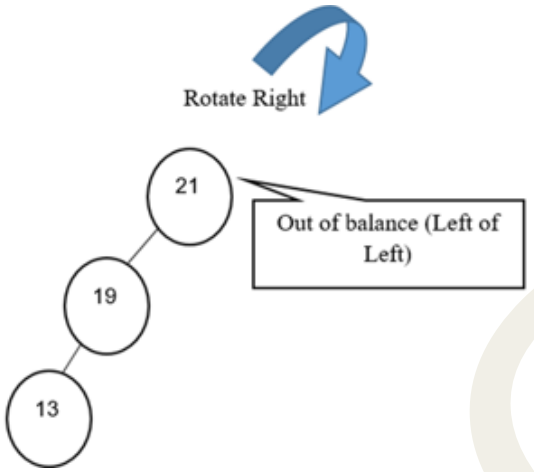


Fig 3.3.16(a) after inserting13

After

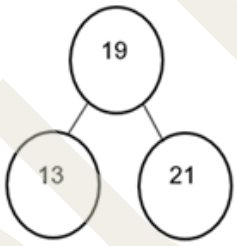


Fig 3.3.16(b) after rotation

The out-of-balance condition created by a left-high subtree of a left-high tree is balanced by rotating the out-of-balance node to the right. In Figure 3.3.16(a), the node 21 tree is out of balance because the left subtree 19 is left high, and it is on the left branch of node 21, which is also left high. In this case, we balance the tree by rotating the root, 21, to the right so that it becomes the right subtree of 19.

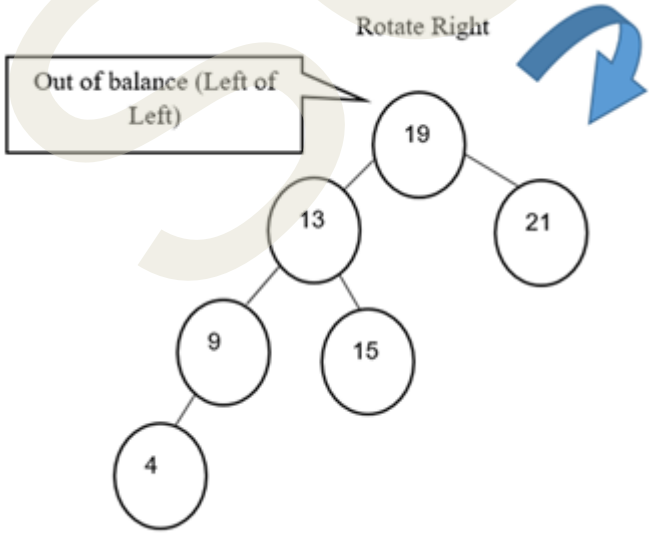


Fig 3.3.17(a) after inserting 4

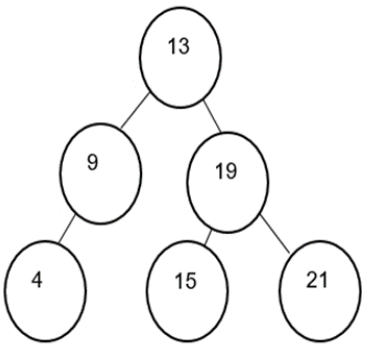


Fig 3.3.17(b) after rotation

Figure 3.3.17(a) is an unbalanced tree. However, the subtree 13 is balanced. Hence, we have to rotate 19 to the right, making it as the right subtree of root 13 (Figure 3.3.17(b). However, when this is done, we are left with node 15, the current right tree of 13. Since the old root, 19, loses its left subtree (13) in the rotation (as 13 becomes the root), we can, therefore, use 19's empty left subtree to attach 15. This also preserves the search tree relationship, which states that all nodes on the right of the root must be greater than or equal to the root.

Case 2: Right of Right Balancing

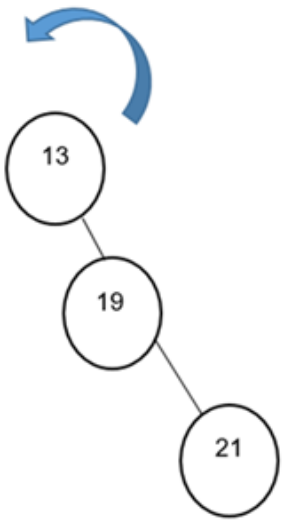


Fig 3.3.18(a)

After

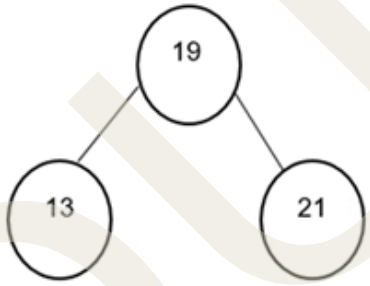


Figure 3.3.18(b)

Case 2 is an example of a simple left rotation. Here, subtree 19 is balanced, but the root is not (Figure 3.3.18(a)). Hence, we rotate the root to the left, making it the left subtree of the new root, 19(Figure 3.3.18(b)).

Case 3: Right of Left Balancing

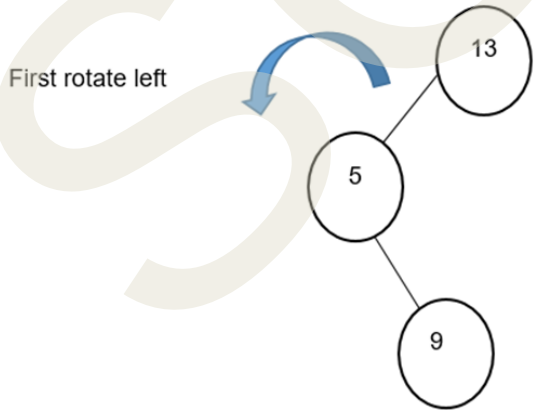


Fig 3.3.19(a)

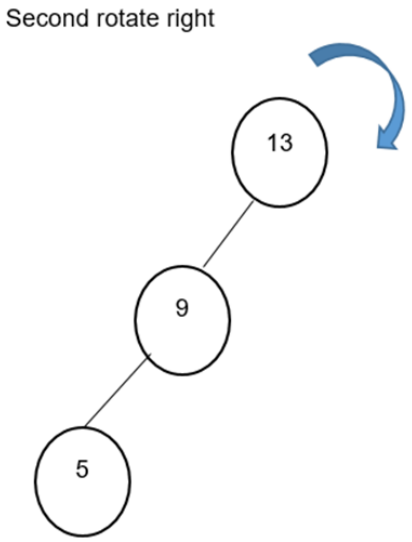


Fig 3.3.19(b)

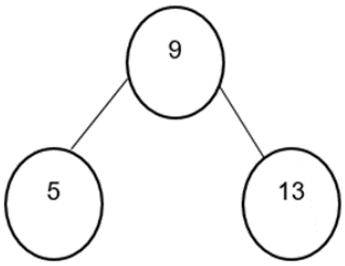


Fig 3.3.19(c)

We see an out-of-balance tree in which the root is left high, and the left subtree is right high a right-of-left tree. To balance this tree, we first rotate the left subtree to the left(-Figure 3.3.19(a), then we rotate the root to the right (Figure 3.3.19(b)), making the left node the new root (Figure 3.3.19(c)).

To balance the tree, we first rotate the right subtree (45) right (Figure 3.3.20(a)) and then rotate the root (13) left (Figure 3.3.20(b)), resulting in Figure 3.3.20(c).

Case 4: Left of Right Balancing

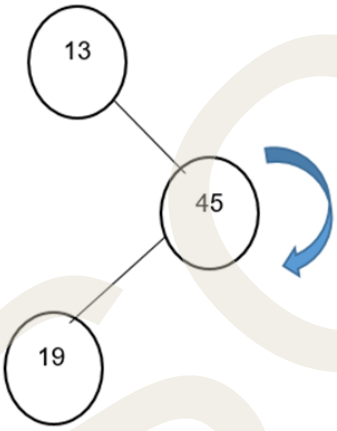


Fig 3.3.20(a)

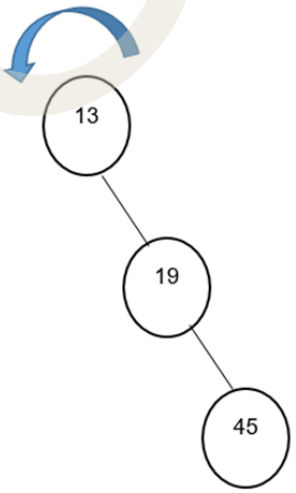


Fig 3.3.20(b)

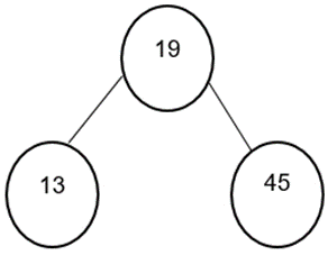


Fig 3.3.20(c)

3.3.4 B-Tree

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalised form of the binary search tree.

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in the main memory. To understand the use of B-Trees, we must think of a huge amount of data that cannot fit in the main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea behind using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting the maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Following are the features of a B-Tree

- All leaves are at the same level.
- A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
- Every node except the root must contain at least t-1 keys. The root may contain a minimum 1 key.
- All nodes (including root) may contain at most 2t – 1 keys.
- The number of children of a node is equal to the number of keys in it plus 1.
- All keys of a node are sorted in increasing order. The child between two keys, k1 and k2, contains all keys in the range from k1 to k2.
- B-tree grows and shrinks from the root, unlike the Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(Logn).

It is also known as a height-balanced m-way tree. The following Figure 3.3.21 shows a B tree. Here, the root node contains more than 1 key value (Here, we have 3 pointers and 2 key values). Also, the root node has more than two children.

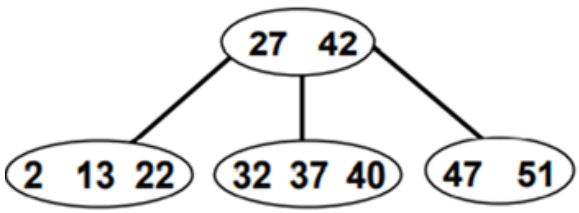


Fig 3.3.21 B –Tree

The need for B-tree arose with the rise in the need for less time in accessing the physical storage media like a hard disk. Secondary storage devices are slower and have a larger capacity.

Search in B Tree is similar to search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has a key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL. Traversal is also similar to the Inorder traversal of Binary Trees. We start from the leftmost child, recursively print the leftmost child, and then repeat the same process for the remaining children and keys. In the end, recursively print the rightmost child.

Inserting into a B-tree involves finding the correct leaf node where the new key should be placed. If this node has space, the key is simply inserted. However, if the node is full, it is split into two nodes, and the middle key is moved up to the parent node, which may recursively cause further splits. Deleting from a B-tree is more complex and depends on the location of the key. If the key is in a leaf node, it can be removed directly. If the key is in an internal node, it must be replaced with either the largest key from its left subtree or the smallest key from its right subtree. The replacement key is then deleted from the leaf node where it originally resided. If a node becomes deficient (i.e., it has fewer keys than the minimum required), it must be rebalanced through borrowing a key from a sibling node or merging with a sibling, which might also necessitate further rebalancing up the tree.

Recap

- The aim of balanced tree - to reach the leaf in a minimum of traversal
- AVL tree got its name after its inventor, Georgy Adelson-Velsky and Landis.
- AVL trees are a special type of binary tree.
- B-tree is a special type of self-balancing search tree
- B-trees contain more than one key and can have more than two children

Objective Type Questions

1. For a node in a binary tree, what is the term used to differentiate between the heights of its left subtree and right subtree?

2. Which type of tree contains more than one key and can have more than two children?

3. A binary tree is balanced if the difference between the left and right subtree of every node is not more than which value?

4. What is the other name for a height-balanced binary tree?

5. Which factor does the search time in a BST depend upon?

6. Which is the highest value of the difference between the heights of the left and right subtrees of all the nodes in a BST that makes the efficiency of searching considered to be ideal?

7. What is the term used to differentiate between the height of the left subtree and the height of the right subtree?

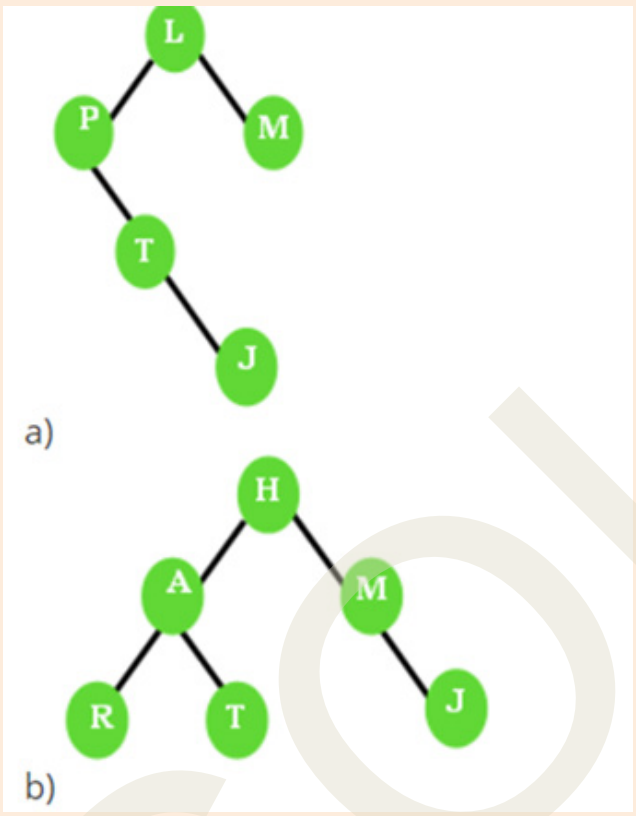
8. What is the name of a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children?

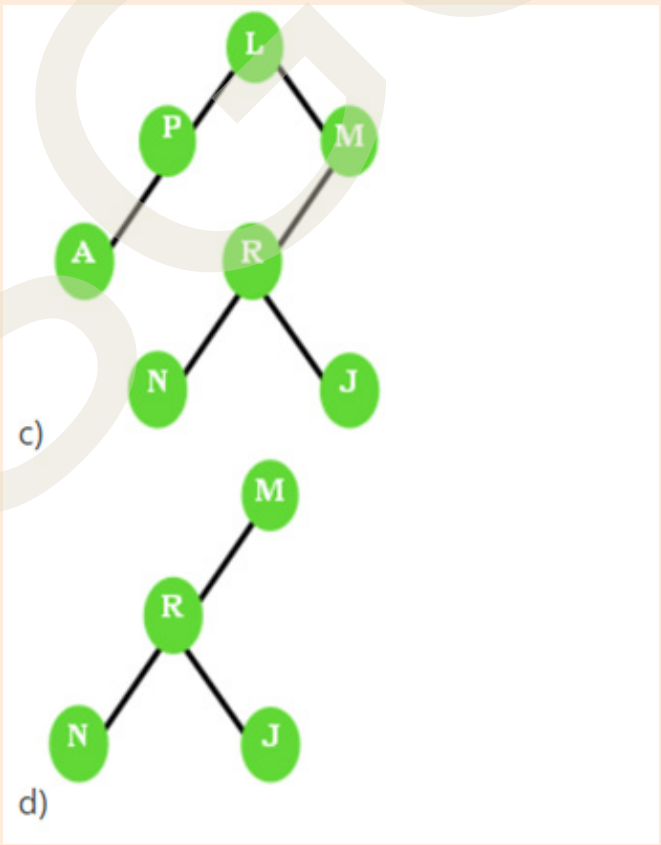
Answers to Objective Type Questions

1. Balance factor
2. B-tree
3. One
4. Balanced binary tree
5. Height
6. One
7. Balancing factor
8. B-tree

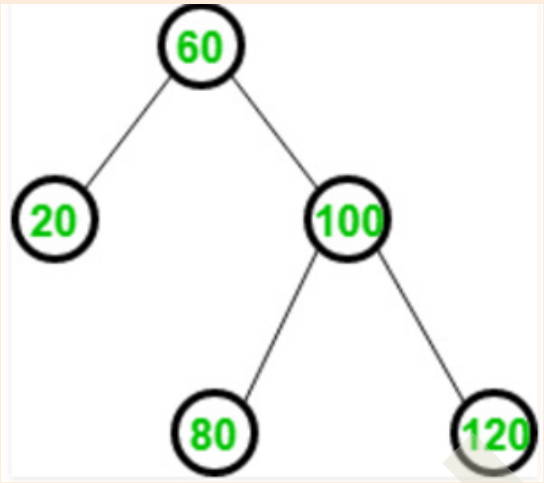
Assignments

1. Identify the balanced tree among the following trees.



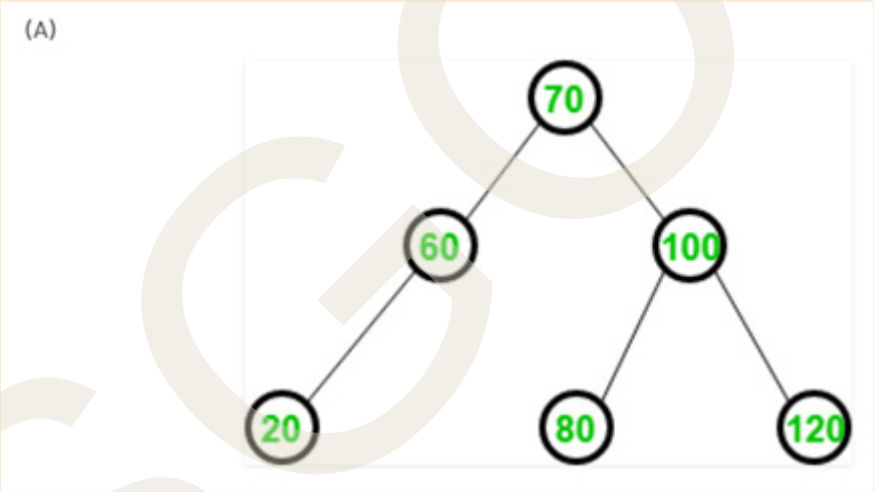


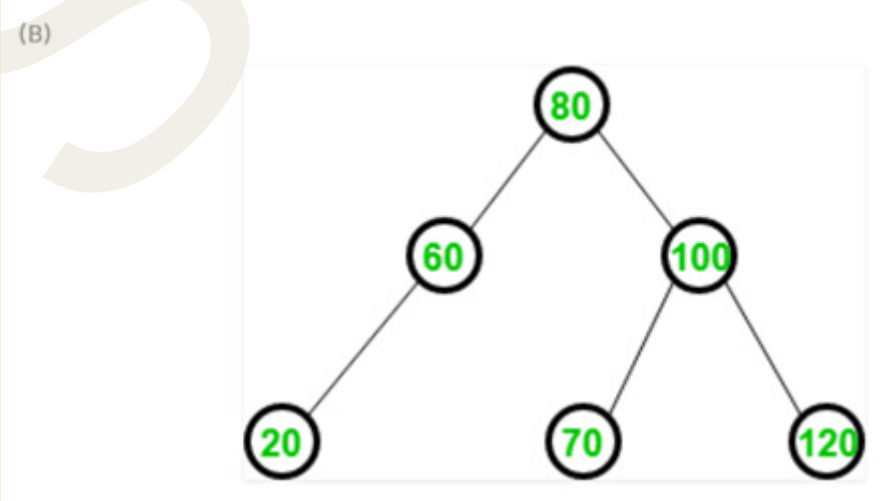
2. Consider the following AVL tree.



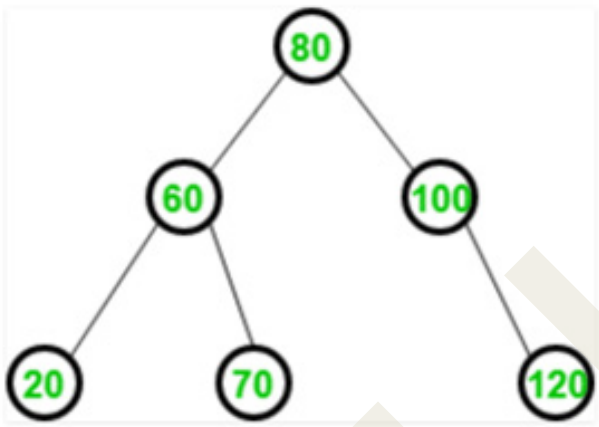
Which one of the following will be the updated AVL tree after the insertion of 70?

3. Which one of the following will be the updated AVL tree after the insertion of 70?





(C)



(D) None

4. Explain B-Tree. What are the advantages of B-Tree over the other trees?
5. Explain different searching techniques in B-Tree.
6. Explain the balanced binary tree and its advantages for searching over the unbalanced tree.
7. Explain the AVL tree.

Suggested Reading

1. Sharma, A. K. Data Structures using C, 2e. Pearson Education India, 2013.
2. Kruse, Robert, and C. L. Tondo. Data structures and program design in C. Pearson Education India, 2007.
3. Mark, Allen Weiss. "Data structures and algorithm analysis in C." (1992).
4. Hopcroft, John E., Jeffrey D. Ullman, and Alfred Vaino Aho. Data structures and algorithms. Vol. 175. Boston, MA, USA: Addison-Wesley, 1983.

UNIT 4 Graphs

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- study different graph types and basic terminologies of graphs
- familiarise the concept of adjacency matrix and adjacency list representation of graphs
- acquire knowledge of multi-list representation of graphs
- learn the implementation of graphs
- understand the concept of graph traversal
- gain awareness of breadth-first search
- obtain the concept of depth-first search
- get an idea of applications of graph data structures

Prerequisites

In the previous block, we studied a tree, which is also a non-linear data structure. There is a hierarchical relationship between parents and children in the tree data structure. That is one parent and many children. However, in a graph, the relationship is less restricted. That is, the relationship is between many parents and many children. Every tree is a graph but not conversely. Graphs are data structures that have wide-ranging applications in real life, like airlines, analysis of electrical circuits, finding shortest routes, flow charts of a program, etc. Many real-world problems can be modelled using a graph. Graphs can be used to represent any collection of objects having some kind of pairwise relationship. Consider a city route map in your area. It may include several roads and traffic junctions.

Let's consider an example route map as shown in Figure 3.4.1 (a). We can see several roads and junctions here. Now, select a particular area from the map. In Figure 3.4.1 (b), we can see the selected area in a rectangle. We can represent junctions as nodes and the roads as edges. Here, six junctions are in the selected area. All the six junctions

are represented using nodes. The roads connecting these junctions are represented by edges. Thus, a small area of the map is represented by a graph, which is shown in Figure 3.4.1 (c).

Figure 3.4.1 Route Map

So, a graph contains a set of nodes or vertices and edges or links. An edge connects two vertices.

Key Concepts

Undirected & Directed Graphs, Weighted Graphs, Regular and Planar Graphs, Cyclic and Acyclic Graphs, Connected and Disconnected Graphs, Biconnected Graphs, Graph Traversal and Applications of Graph.

Discussion

3.4.1 Definition of Graph

A graph G consists of a non-empty finite set of vertices V(G) and a finite set of edges E(G), where V(G)=$(v_0, v_1, ..., v_n)$ or set of vertices or nodes.

And

$E(G) = (e_1, e_2, ..., e_n)$ or set of edges.

The set of edges contains a pair of vertices. If $e = (v_i, v_j)$ is an edge with vertices v_i and v_j, and $v_i, v_j \in V(G)$, then v_i and v_j are said to lie on edge, e and e is said to be incident with v_i and v_j.

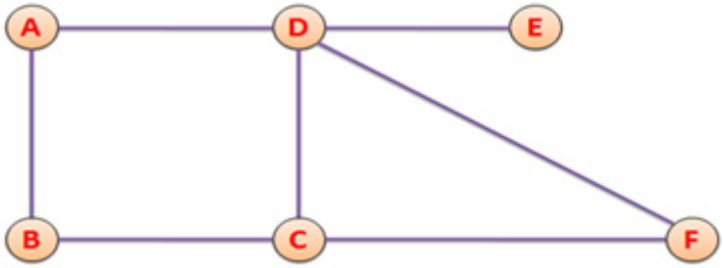


Fig 3.4.2 Example of simple Graph

Consider a simple graph ***G*** with vertices A, B, C, D, E, and F, as shown in Figure 3.4.2. We can see that there are 6 vertices or nodes and 7 edges. That is,

V (G) = { A, B, C, D, E, F }

E (G) = { (A, B), (A, D), (B, C), (D, C), (C, F), (D, E), (D, F) }

In the edge set ***E (G)***,

- (A, B) represents the edge between the nodes A, B.
- (A, D) represents the edge between the nodes A, D.
- (B, C) represents the edge between the nodes B, C.
- (D, C) represents the edge between the nodes D, C.
- (C, F) represents the edge between the nodes C, F.
- (D, E) represents the edge between the nodes D, E.
- (D, F) represents the edge between the nodes D, F.

Problem 1: Draw a simple graph with 5 vertices and 4 edges whose vertices are P, Q, R, S, and T. The edges of the graph are PQ, QR, PS, and ST.

Solution: First, represent 5 vertices P, Q, R, S, and T with 5 different nodes. To represent a node, we use small circles. Each circle is labelled with a name: P, Q, R, S, and T are the labels. In the next step, add each edge by connecting the corresponding vertices. Take the first edge, PQ. The vertices are P and Q, so connect node P and node Q with a line. Then add the next edge, QR, followed by PS, and finally ST. This completes the graph as shown in Fig 3.4.3.

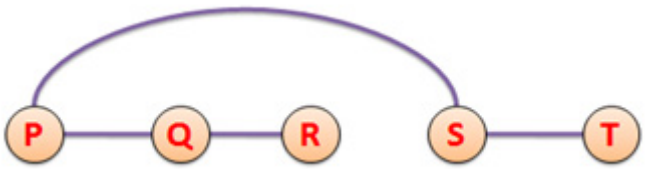


Fig 3.4.3 The resultant graph with 5 vertices and 4 edges

3.4.2 Types of Graphs

A graph can be broadly classified into two types: Undirected graph and Directed graph.

Undirected Graph:

If the pair of vertices is unordered, then graph ***G*** is called an undirected graph. This means if there is an edge between v_1 and v_2, it can be represented as (v_1, v_2) or (v_2, v_1). Fig. 3.4.4 shows an example of an undirected graph. It consists of 5 vertices and 5 edges. That is,

V (G) = { A, B, C, D, E }

E (G) = { (A, B), (A, C), (A, D), (D, C), (C, E) }

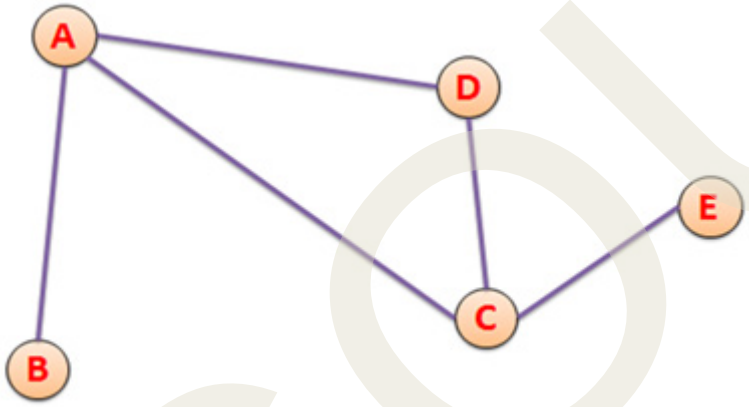


Fig 3.4.4 Example of Undirected Graph

Here, the edge (A, B) can also be represented as (B, A). The edge (A, C) can be represented as (C, A). Similarly, the other 3 edges can be represented as (D, A), (C, D), and (E, C). The pairs of vertices are unordered. So, we can call it an undirected graph.

Directed Graph:

If the pair of vertices are ordered, then the graph G is called a directed graph or a digraph. This means that a graph has an ordered pair of vertices (v_1, v_2), where v_1 is the tail and v_2 is the head of the edge; we can call it a digraph. The line segments or arcs (edges) of the directed graphs have arrowheads which indicate the direction. Figure 3.4.5 shows an example of a directed graph. It has 5 vertices and 6 edges.

That is,

V (G) = { A, B, C, D, E }

E (G) = { (A, D), (D, E), (E, C), (C, D), (B, C), (B, A) }

Here, we can see that the edge (A, D) is represented with an arrow-headed line whose tail is A and the head of the edge is D. We can't represent the same edge as (D, A) because the vertex D is not the tail and the vertex A is not the head of the edge. Like-wise, all other edges are represented in this manner in the given directed graph.

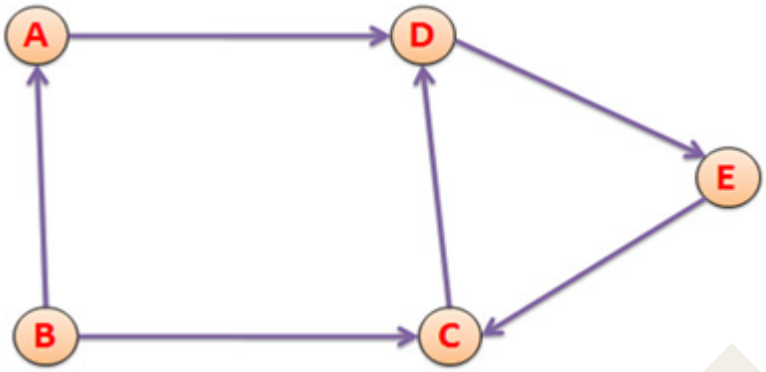


Fig 3.4.5 Example of directed Graph

3.4.3 Terminologies Used in Graphs

3.4.3.1 Weighted Graph

Consider a city route map represented by a graph. The junctions are represented using nodes and the roads connecting them are represented using edges. Take an edge connecting two nodes (vertices). We can label this edge with a value that corresponds to the distance between those nodes. We can say it as the weight of that edge. Similarly, the speed limit on a road (edge in the graph) can be represented as weight.

If all the edges in a graph are labelled with some numbers or weights, then the graph is called a weighted graph. Figure 3.4.6 shows an example of a weighted graph. Here, P, Q, R, and S are the vertices of the graph. The graph consists of 4 edges, and all are labelled with some weights. Edge PQ is labelled with a weight 4. Edge PR is labelled with a weight 2. Edge RS is labelled with a weight 5, and edge QS is labelled with a weight 3.

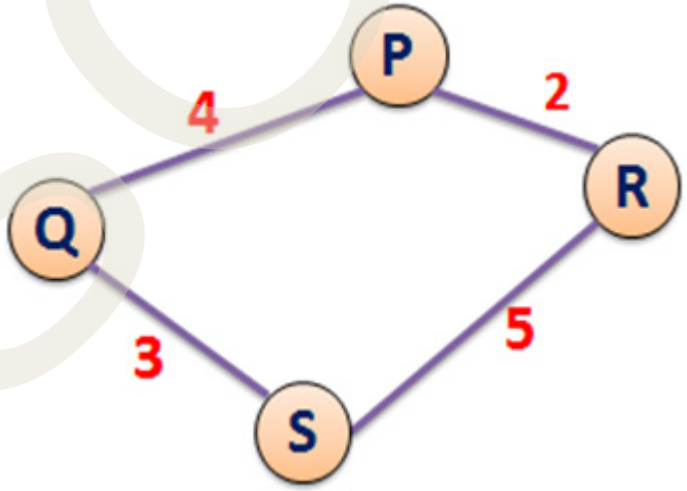


Fig 3.4.6 An example of a Weighted Graph

3.4.3.2 Self Loop

For a graph G, if there is an edge whose starting and ending vertices are the same, that is (v1, v1), then it is called a self-loop. Figure 3.4.7 shows an example of a graph with a

self-loop. Here v1, v2, v3, and v4 are the vertices of the graph. The edges are (v1, v2), (v2, v3), (v3, v4), (v4, v1) and (v1, v1) and the edge (v1, v1) is a self loop.

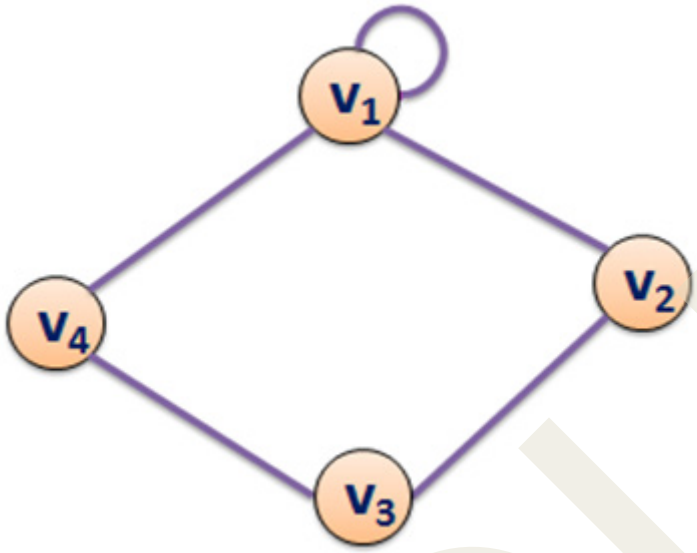


Fig 3.4.7 An example of Self Loop

3.4.3.3 Parallel Edges

For graph G, if there is more than one edge between the same pair of vertices, then it is known as a parallel edge. Figure 3.4.8 shows a graph with parallel edges. We can see there are two edges between the same pair of vertices, v1 and v2.

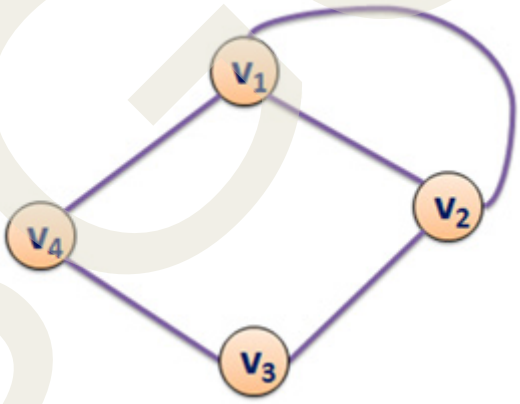


Fig 3.4.8 An example for Parallel Edges

3.4.3.4 Adjacent Vertices

In graph G, a vertex u is adjacent to another vertex v if there is an edge from u to v. In an undirected graph, if (v1, v2) is an edge, then v1 is adjacent to v2, and v2 is adjacent to v1. In a directed graph, if (v1, v2) is an edge, then v1 is adjacent to v2, and v2 is adjacent to v1. For example, Figure 3.4.9 (i) shows an undirected graph; vertex u is adjacent to vertex v, and vertex v is adjacent to vertex u. Figure 3.4.9 (ii) shows a directed graph; vertex u is adjacent to vertex v, and vertex v is adjacent to vertex u.

Fig 3.4.9 An example for adjacent vertices and Incidence

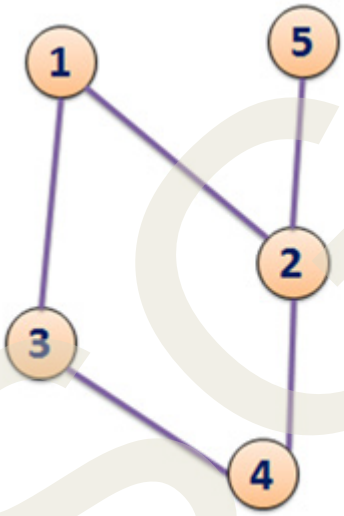
3.4.3.5 Incidence

In an undirected graph, the edge (u, v) is incident on vertices u and v. In a directed graph, the edge (u, v) is incident from node u and incident to node v.

For example, Figure 3.4.9(i) shows an undirected graph; the edge (u, v) is incident on vertices u and v. Figure 3.4.9(ii) shows a directed graph; edge (u, v) is incident from node u and is incident to node v.

3.4.3.6 Degree of Vertex

The degree of a vertex is the number of edges incident to that vertex. The degree of a node in an undirected graph is the number of edges connected to that node.



Vertex	Degree of Vertex
1	2
2	3
3	2
4	2
5	1

Fig 3.4.10 Degree of Vertex in an Undirected Graph

Take an example graph shown in Figure 3.4.10 to find the degree of a vertex. Here, the undirected graph contains five vertices. We can find the degree of each vertex in Figure 3.4.10.

The **degree of vertex 1** is **2** because 2 edges are incident to vertex 1.

The **degree of vertex 2** is **3** because 3 edges are incident to vertex 2.

The **degree of vertex 3** is **2** because 2 edges are incident to vertex 2.

The **degree of vertex 4** is **2** because 2 edges are incident to vertex 4.

The **degree of vertex 5** is **1** because only 1 edge is incident to the vertex 5.

In a directed graph, there are two types of degrees for every node: in-degree and out-degree.

In degree: The in degree of a vertex is the number of edges coming to that vertex or edges incident to it.

Out degree: The out degree of a vertex is the number of edges going outside from that node or the edges incident from it.

Take an example digraph shown in Fig. 3.4.11 to find the degree of a vertex. Here, the directed graph contains five vertices. We can find the in degree and out degree of each vertex in Figure 3.4.11.

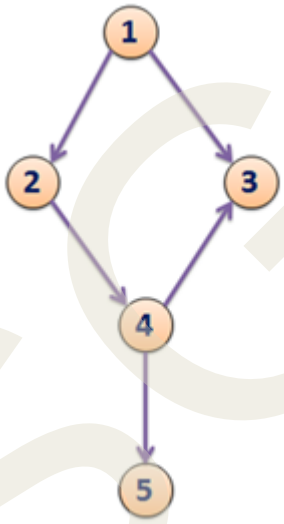
The **in degree** of vertex 1 is **0** because no edges are incident to vertex 1.

The **in degree** of vertex 2 is **1** because only 1 edge is incident to the vertex 2.

The **in degree** of vertex 3 is **2** because 2 edges are incident to vertex 3.

The **in degree** of vertex 4 is **1** because only 1 edge is incident to the vertex 4.

The **in degree** of vertex 5 is **1** because only 1 edge is incident to the vertex 5.



Vertex	In degree	Out Degree
1	0	2
2	1	1
3	2	0
4	1	1
5	1	0

Fig 3.4.11 Degree of Vertex in a Directed Graph

The **out degree** of vertex 1 is **2** because 2 edges are incident from the vertex 1.

The **out degree** of vertex 2 is **1** because only 1 edge is going outside from the vertex 2.

The **out degree** of vertex 3 is **0** because no edges are going outside from the vertex 3.

The **out degree** of vertex 4 is **1** because only 1 edge is incident from the vertex 4.

The **out degree** of vertex 5 is **0** because no edges are going outside from the vertex 5.

3.4.3.7 Simple graph

A graph or digraph that does not only have self-loop or parallel edges is called a simple graph. Figure 3.4.12 shows examples of simple graphs. Fig(i) shows a simple undirected graph. There are no self-loops and parallel edges. Fig(ii) shows a simple digraph with no self-loops and parallel edges.

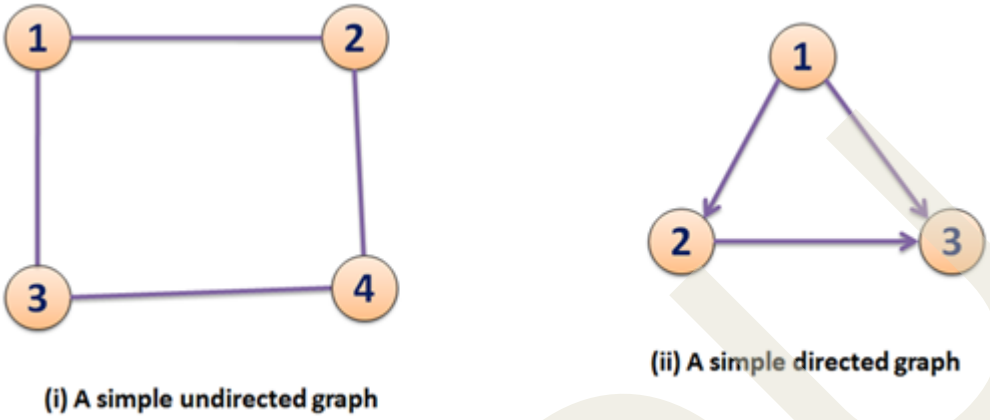


Fig 3.4.12 Examples of Simple Graph

3.4.3.8 Multi Graph

A graph which has either a self-loop or parallel edges or both is called a multi-graph.

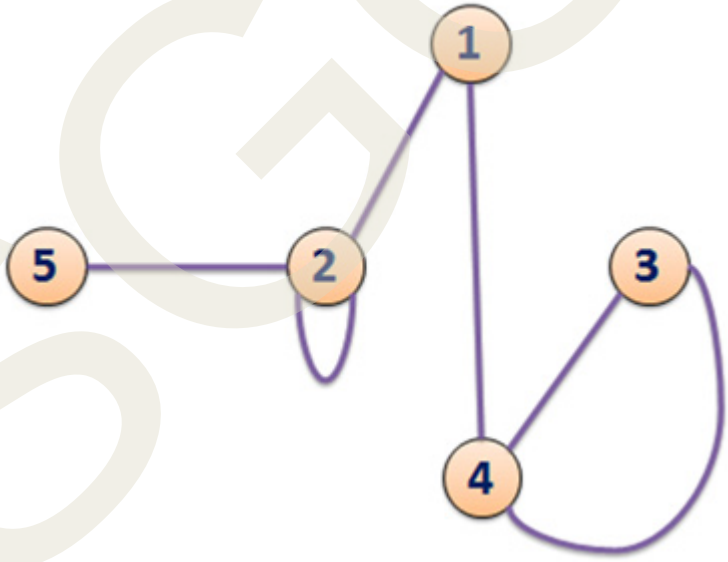


Fig 3.4.13 Example for Multi-Graph

Figure 3.4.13 shows an example of a multi graph. The graph consists of 5 vertices and 6 edges. There is a self loop in node 2. The vertices 4 and 3 are connected with 2 parallel edges. So, it is a multi graph.

3.4.3.9 Maximum Edges in Graph

In a simple undirected graph, there can be n(n–1)/2 maximum edges, and in a simple

directed graph, there can be n(n–1) maximum edges. Where n is the total number of vertices in the graph.

Fig 3.4.14 Examples of Maximum edges

For example refer (Figure 3.4.14 (i)), if the number of nodes of a simple undirected graph is 2, then the maximum number of edges will be 1. That is $(2 \times (2-1))/2 = 1$.

For example refer (Figure 3.4.14(ii)), if the number of nodes of a simple directed graph is 2, then the maximum number of edges will be $(2 \times 1) = 2$.

3.4.3.10 Complete Graph

A graph is said to be complete if each vertex is adjacent to every other vertex in the graph.

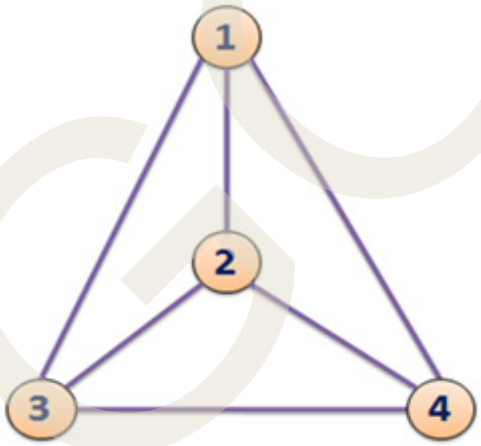


Fig 3.4.15 Example of Complete Graph

Figure 3.4.15 shows an example of a complete graph. In the figure, we can see four vertices: say 1, 2, 3, and 4. Vertex 1 is adjacent to vertices 2, 3, and 4. Vertex 2 is adjacent to vertices 1, 3, and 4. Vertex 3 is adjacent to vertices 1, 2, and 4. Vertex 4 is adjacent to vertices 1, 2, and 3. So, each vertex is adjacent to every other vertex in the graph. So, we can call it a complete graph. The total number of edges in an undirected complete graph will be n(n–1)/2, where n is the total number of vertices or nodes.

In Figure 3.4.15, n = 4. So the total number of edges will be (n*(n-1))/2; ie. 12/2 = 6 edges.

3.4.3.11 Regular Graph

A graph is regular if every node is adjacent to the same number of nodes. That is, each node in the graph has the same degree. Figure 3.4.16 shows an example of a regular

graph. It consists of 10 nodes. Each node is adjacent to the same number of nodes. That is, each node has the same degree. Take node 1. It is adjacent to nodes 2, 3 and 6. Therefore, the degree of node 1 is 3. Node 2 is adjacent to nodes 1, 4 and 7. The degree of node 2 is 3.

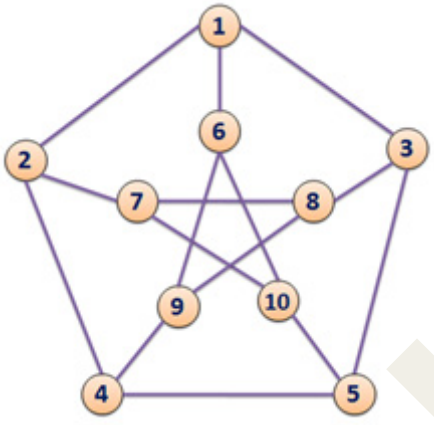


Fig 3.4.16 Example of Regular Graph

Likewise, we can see that all other nodes in the graph have the same degree. That is, the degree of nodes 3, 4, 5, 6, 7, 8, 9 and 10 is equal to 3. So, it is a regular graph.

3.4.3.12 Planar Graph

A graph is planar if it can be drawn in a plane without any two edges intersecting.

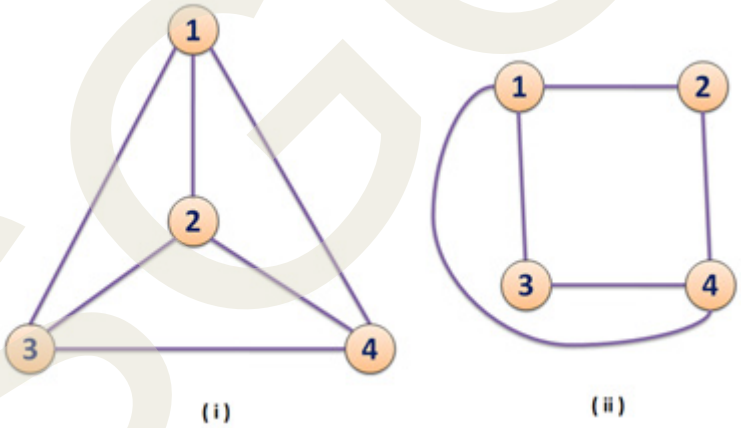


Fig 3.4.17 Examples of Planar Graphs

Figure 3.4.17 shows examples of planar graphs. In both graphs, no two edges intersect.

3.4.3.13 Walk & Path

In graph ***G***., a **walk** is a finite sequence of edges of the form $\mathbf{v_0v_1}$, $\mathbf{v_1v_2}$, $\mathbf{v_2v_3}$, ……., $\mathbf{v_{n-1}v_n}$. We can also denote this by $\mathbf{v_0} \rightarrow \mathbf{v_1} \rightarrow \mathbf{v_2} \rightarrow \mathbf{v_3} \rightarrow \ldots\ldots \rightarrow \mathbf{v_n}$ in which any 2 consecutive edges are adjacent or identical. Such a walk determines a sequence of vertices $\mathbf{v_0}$, $\mathbf{v_1}$, $\mathbf{v_2}$, ……., $\mathbf{v_n}$. We can call $\mathbf{v_0}$ as the initial vertex and $\mathbf{v_n}$ as the final vertex of the walk. That is a walk from $\mathbf{v_0}$ to $\mathbf{v_n}$.

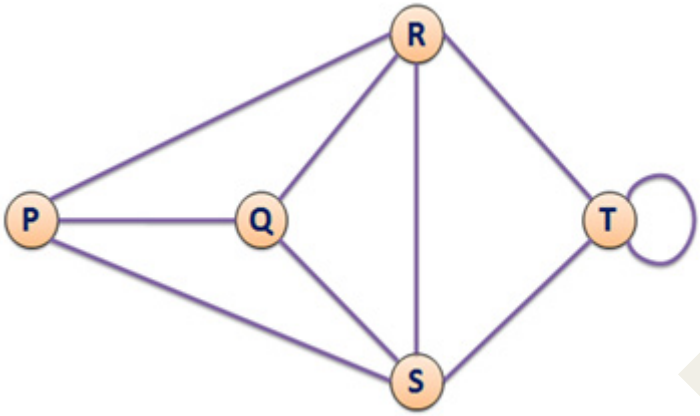


Fig 3.4.18 Example of walk and path

The number of edges in a walk is called its length. For example, in Figure 3.4.18, **P → Q → R → S → T → T → S → Q** is a walk of length 7 from P to Q.

If all the edges in a walk are distinct, then it is called a trail. If all the vertices in a trail are distinct, then it is called a path. That is, if the vertices in a trail, v_0, v_1, v_2, ….., $v_{n,}$ are distinct, then the trail is a path. If $v_0 = v_{n,}$ then the path or trail is called a closed path or a closed trail.

Consider Figure 3.4.18. We can see that

P → Q → R → S → T → T → R is a trial.

P → Q → R → S → T is a path.

P → Q → R → S → T → R → P is a closed trail.

P → Q → S → T → R → P is a closed path.

3.4.3.14 Cycle

If there is a path containing one or more edges that start from a vertex and terminate at the same vertex, then the path is called a cycle. A closed path containing at least one edge is a cycle. Any loop or pair of multiple edges is a cycle.

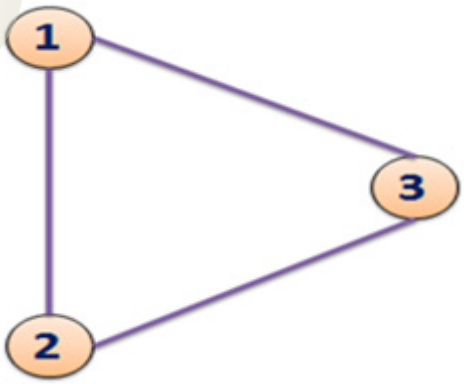


Fig 3.4.19 Examples of a Cycle

Figure 3.4.19 shows an example of a cycle. Take vertex 1, and we can see path **1** →**2** → **3** → **1**. That is a path starting from 1 and terminating at 1. So it is a cycle.

3.4.3.15 Cyclic Graph

A graph that has cycles is called a cyclic graph. In Figure 3.4.20, we can see both directed and undirected graphs with a cycle.

The 1st digraph is cyclic because it contains a cycle **1** → **2** → **3** → **1.**

The 2nd undirected graph is also cyclic because it contains a cycle **1**→ **2** → **3** → **4** → **1.**

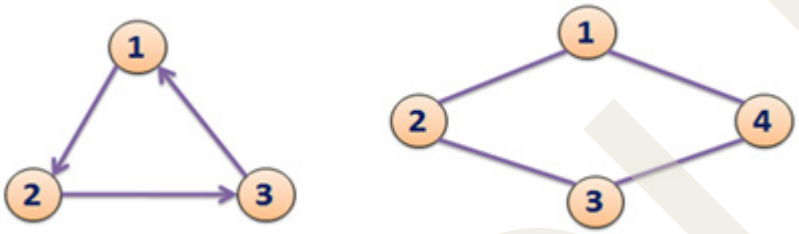


Fig 3.4.20 Examples of Cyclic Graph

3.4.3.16 Acyclic Graph

If a graph does not have any cycle, then it is called an acyclic graph. In Figure 3.4.21, we can see both directed and undirected graphs with no cycles. The first digraph is acyclic because it contains no cycle. It contains the following paths;

1 → 2 → 3, 1 → 3, 2 → 3

All the three paths are not closed. So there is no cycle. That is, the digraph is acyclic.

Fig 3.4.21 Examples of Acyclic Graph

The second undirected graph, shown in Figure 3.4.21, is also acyclic because it contains no cycle. It contains the following paths;

1 → 2 → 3, 3 → 2 → 1, 2 →1, 1 → 2, 2 → 3, 3 → 2

All the six paths are not a cycle. So, the graph is acyclic.

3.4.3.17 Connected Graph

In graph G, two vertices, v1 and v2, are said to be connected if there is a path in G from

v1 to v2 or v2 to v1. A graph is said to be connected if there is a path from any node of the graph to any other node. That is, a path exists for every pair of distinct vertices in G. Figure 3.4.22 (i) shows a connected graph. Here, we can see that each pair of nodes is connected. There is a path from node 1 to nodes 2, 3 and 4. Likewise, all other nodes have a path to every other node. Figure 3.4.22 (ii) shows a disconnected graph. The graph is disconnected because there is no path to node 4 from the nodes 1, 2 and 3.

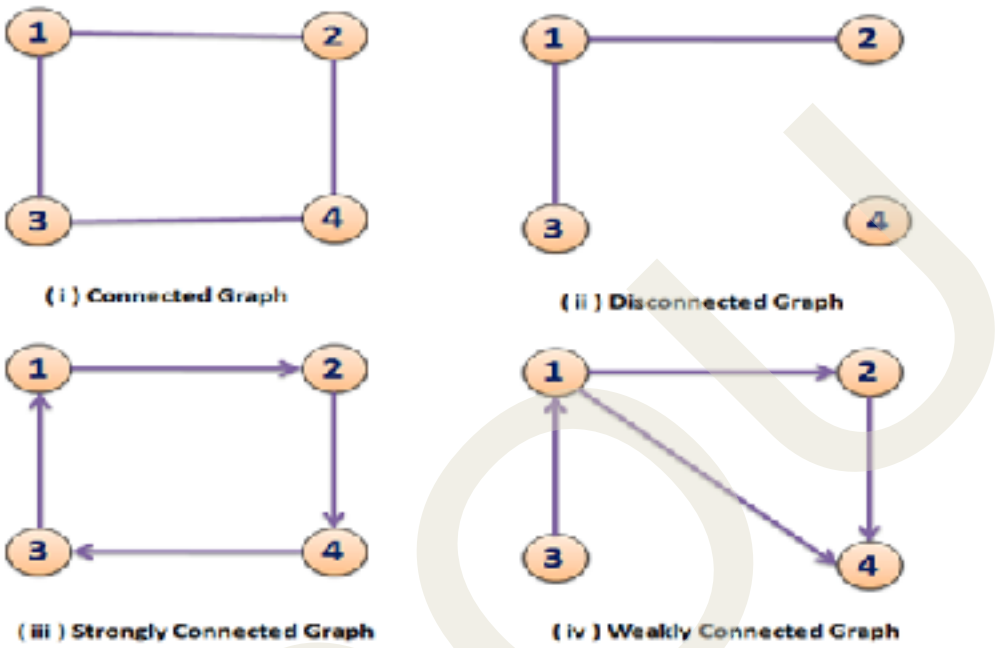


Fig 3.4.22 Examples of Connected and Disconnected Graphs

A directed graph ***G*** is said to be **strongly connected** if there is a path for every pair of distinct vertices in G. Figure 3.4.22 (iii) shows a strongly connected digraph. For example, take the case of vertex 1; there is a path from 1 to 2, 1 to 3, and 1 to 4. In the case of vertex 2, there is a path from 2 to 3, 2 to 4, and 2 to 1. Likewise, in the case of vertex 3 and vertex 4, there is a path to all other vertices.

A digraph is called **weakly connected** if, for any pair of vertices ***u*** and ***v***, there is a path from ***u*** to ***v*** or a path from ***v*** to ***u***. In a digraph, if we replace the directed edge with undirected edges and the resulting graph is connected then that digraph is weakly connected. Figure 3.4.22 (iv) shows a weakly connected graph. In the case of vertex 1, there is a path from 1 to vertices 2 and 4 and a path from vertex 3 to 1. If we replace all the directed edges with undirected edges here, then the resultant graph is a connected one. That is, there is a path from vertex 1 to vertices 2, 3, and 4. There is a path from vertex 2 to vertices 1, 3, and 4. There is a path from vertex 3 to vertices 1, 2, and 4. There is a path from vertex 4 to vertices 1, 2, and 3. So, the graph is weakly connected.

3.4.3.18 Articulation Point

If the graph becomes disconnected on removing a node from the graph, then that node is called the articulation point.

In Figure 3.4.23 (i), node 3 is an articulation point because when node 3 is removed, the

graph becomes disconnected, as shown in Figure 3.4.23 (ii). When node 3 is removed from the graph, all the 5 edges connected to node 3 are removed. That is, edges 3-1, 3-2, 3-6, 3-5 and 3-4 are removed. Then, the resultant graph becomes disconnected.

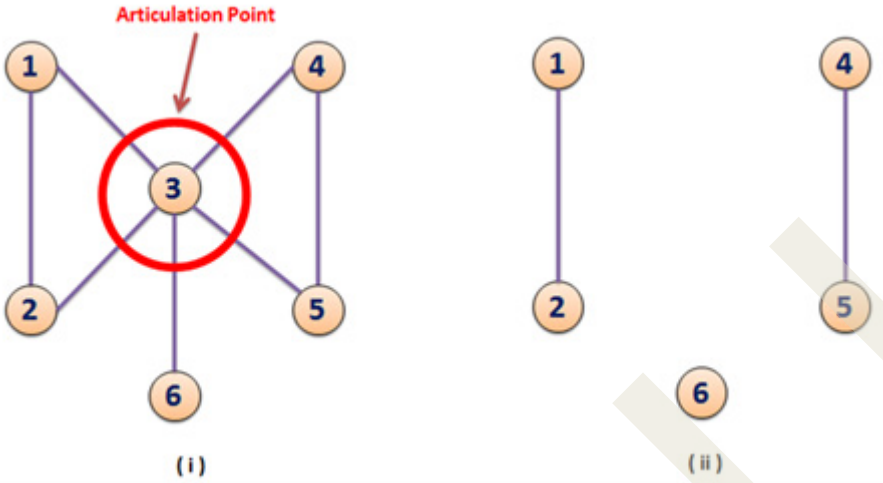


Fig 3.4.23 Example of Articulation point in a graph

3.4.3.19 Bridge

If, on removing an edge from the graph, the graph becomes disconnected, then that edge is called the bridge.

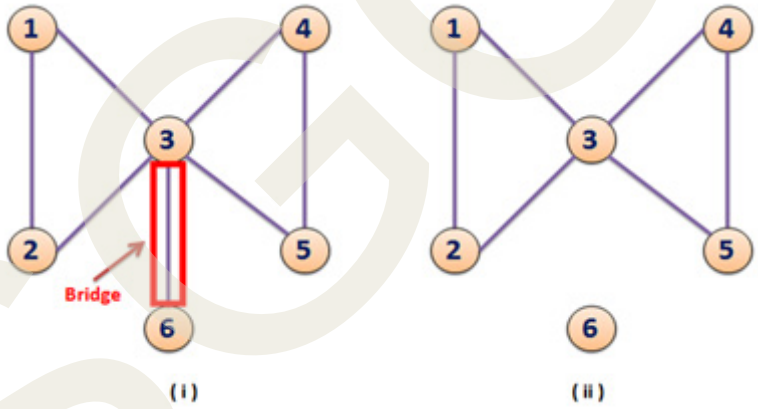


Fig 3.4.24 Example of Bridge in a graph

In Figure 3.4.24 (i), edges 3-6 are a bridge because if this edge is removed, then the graph becomes disconnected, which is shown in Figure 3.4.24 (ii).

3.4.3.20 Biconnected Graph

A graph with no articulation points is called a biconnected graph.

In Figure 3.4.25, there is no articulation point. There are three vertices and three edges in the given graph. Try to remove vertex 1 from the graph, which does not disconnect the graph.

Removing vertex 2 does not disconnect the graph. Removing vertex 3 does not discon-

nect the graph. So, it is a biconnected graph.

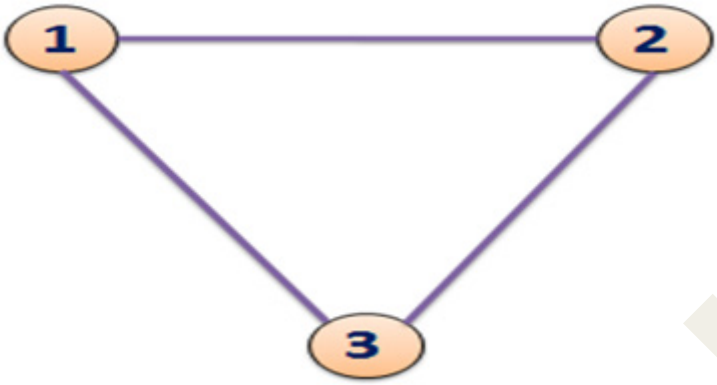


Fig 3.4.25 Example of Biconnected Graph

3.4.4 Graph Representation

3.4.4.1 Adjacency Matrix Representation of Graph

An adjacency matrix is the matrix that keeps the information of adjacent nodes. That is, the matrix keeps the information on whether the vertex is adjacent to any other vertex or not. It is a sequential representation method. The general representation of the adjacency matrix is shown in Figure 3.4.26.

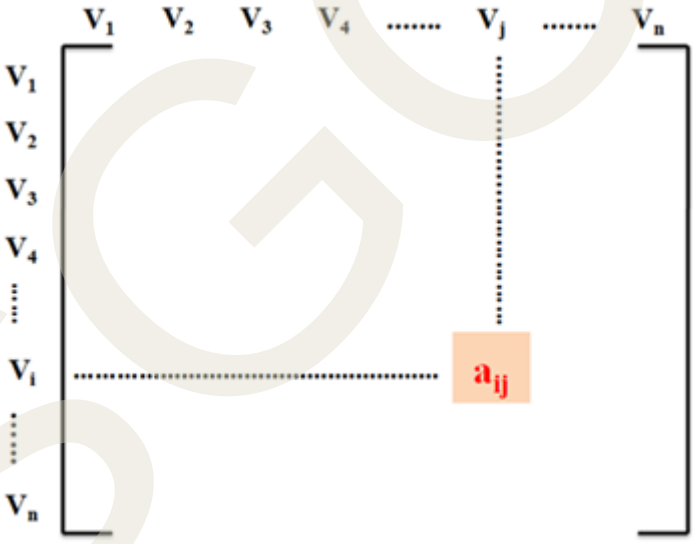


Fig 3.4.26 Adjacency Matrix Representation

Here, V1, V2, V3, ……, Vn are the vertices of the given graph, and aij represents the edge from Vi to Vj. The value of aij is either 1 or 0 according to the rule given below.

aij = 1; if there is an edge from Vi to Vj.

0; otherwise

This adjacency matrix is also called a Bit matrix or Boolean matrix because the entries are either 0 or 1.

Example 1: Consider the following digraph in Figure 3.4.27. From this, we can write the corresponding adjacency matrix.

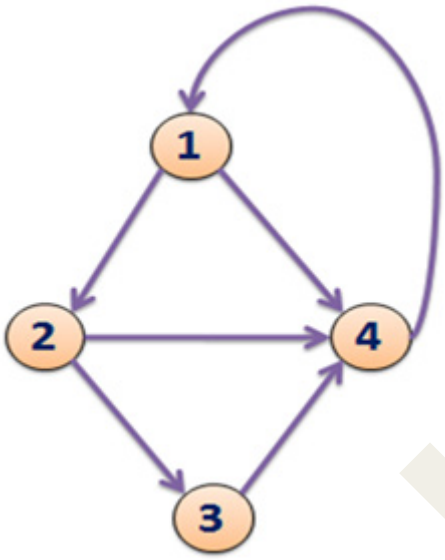


Fig 3.4.27 Example 1

There are 6 directed edges in this graph. First, take the vertex 1. There is an edge from vertex 1 to 2. So we can represent $a_{12} = 1$. There is an edge from vertex 1 to 4. That is $a_{14} = 1$. In the case of vertex 2, there is an edge from 2 to 3 and 2 to 4. That is $a_{23} = 1$, and $a_{24} = 1$. In the case of vertex 3, there is an edge from 3 to 4. That is $a_{34} = 1$. In the case of vertex 4, there is an edge from 4 to 1. That is $a_{41} = 1$. All the other entries of the adjacency matrix will be 0. Then, we can represent the adjacency matrix as;

1 2 3 4 0 1 0 1 0 0 1 1 0 0 0 1 1 0 0 0

Example 2: Consider the following undirected graph in Figure 3.4.28. From this, we can write the corresponding adjacency matrix. There are 6 edges in this graph.

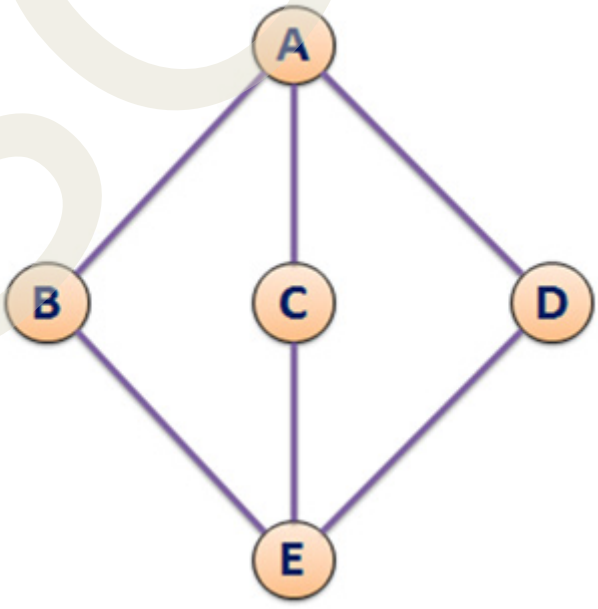


Fig 3.4.28 Example 2

First, take the vertex A. There is an edge from vertex A to B. So, we can represent a_{AB} = 1. There is an edge from vertex A to C and A to D. That is a_{AC} = 1, and a_{AD} = 1. In the case of vertex B, there is an edge from B to E and B to A. That is a_{BE} = 1, and a_{BA} = 1. In the case of vertex C, there is an edge from C to A and C to E. That is a_{CA} = 1 and a_{CE} = 1. In the case of vertex D, there is an edge from D to A and D to E. That is, a_{DA} = 1 and a_{DE} = 1. In the case of vertex E, there is an edge from E to B, E to C, and E to D. So a_{EB} = 1, a_{EC} = 1 and a_{ED} = 1. All the other entries of the adjacency matrix will be 0. Then, we can represent the adjacency matrix as;

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	1
C	1	0	0	0	1
D	1	0	0	0	1
E	0	1	1	1	0

Note:

- The adjacency matrix of an undirected graph is symmetric but not conversely. If A is the adjacency matrix of a simple undirected graph, then $A = A^T$, where A^T is the transpose of A.
- From the adjacency matrix of an undirected graph, the degree of any vertex i is its row sum.
- From the adjacency matrix of a directed graph, the out degree of any vertex i is its row sum and in degree is its column sum.
- The space needed to represent a graph using its adjacency matrix is n^2 bits.
- In the case of multi-graphs, instead of entry 1, the entry will be the number of edges between two vertices.
- For a weighted graph, the entries in the adjacency matrix are the weights of the edges between the vertices.

Example 3: Consider the following weighted digraph in Figure 3.4.29. From this, we can write the corresponding adjacency matrix. There are 5 directed edges in this graph.

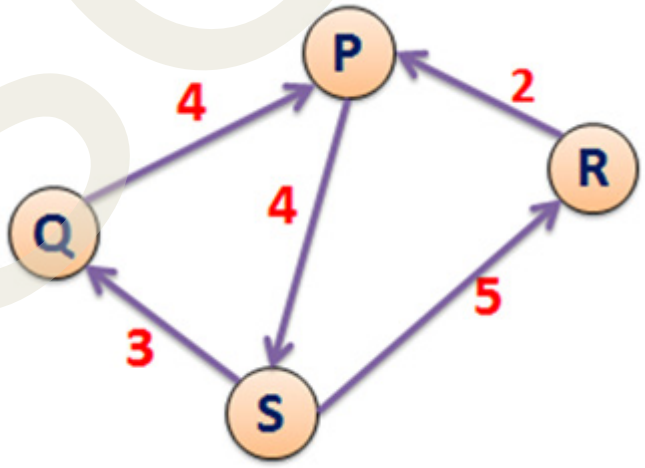


Fig 3.4.29 Example 3

First, take the vertex P. There is an edge from vertex P to S. So we can represent a_{PS} = 4

because the weight of the edge PS is 4. In the case of vertex Q, there is an edge from Q to P. That is a_{QP} = 4 because the weight of the edge QS is 4. In the case of vertex R, there is an edge from R to P. That is a_{RP} = 2 because the weight of the edge RP is 2. In the case of vertex S, there is an edge from S to Q and S to R. That is a_{SQ} = 3 and a_{SR} = 5 because the weight of the edge SQ is 3 and the weight of the edge SR is 5. All the other entries of the adjacency matrix will be 0. Then, we can represent the adjacency matrix as;

P Q R S 0 0 0 4 4 0 0 0 2 0 0 0 0 3 5 0

Example 4: Draw the undirected graph corresponding to the adjacency matrix.

A (G) = 0 1 1 0 1 0 1 1 1 1 0 0 0 1 0 0

Solution: ***A (G)*** is a 4 × 4 matrix, hence G will have 4 vertices. We can take it as 1, 2, 3 and 4. Draw an edge from v_i to v_j where a_{ij}=1. Here;

a_{12}=1 and a_{21}=1. So, draw an undirected edge from vertex 1 to vertex 2.

a_{13}=1 and a_{31}=1. So, draw an undirected edge from vertex 1 to vertex 3.

a_{23}=1 and a_{32}=1. So, draw an undirected edge from vertex 2 to vertex 3.

a_{24}=1 and a_{42}=1. So, draw an undirected edge from vertex 2 to vertex 4.

The resultant graph is shown in Figure 3.4.30.

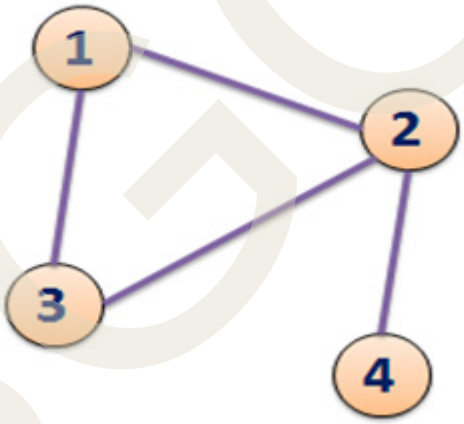


Fig 3.4.30 Example 4

3.4.5 Adjacency List Representation Of Graph

Adjacency matrix representation is a sequential representation method. For a graph G with n vertices or nodes, it may be difficult to insert and delete nodes with sequential representation in memory. This is because the size of the matrix may need to be changed, and the nodes may need to be reordered. So, we need a better representation of G in memory. For this, we use linked representation. It is also known as adjacency list representation. In adjacency list representation, the number of lists depends on the number of vertices. For example, if there are 5 nodes in the graph, then in adjacency list representation consists of 5 separate lists. That is, the adjacency list represents a graph as an array of linked lists. The index of the array represents the number of vertices.

Each element in the list represents the other vertices that make an edge with the vertex. The following session explains how the graphs (both undirected and directed) are represented with an adjacency list with suitable diagrams.

3.4.5.1 Representing an Undirected Graph

Consider an example undirected graph shown in Figure 3.4.31(i). It has four vertices. So, the linked representation includes 4 separate lists. We can see the adjacency list representation in Figure 3.4.31(ii). The index of the array starts with 1. So, the first list represents the adjacent vertices of vertex 1. The adjacent vertices of vertex 1 are 2, 3, and 4. So, the list contains four node structures. It starts from 1, then points to 2, then points to 3, then points to 4. 4 is the terminal node, which means the adjacency list of vertex 1 terminates at node 4.

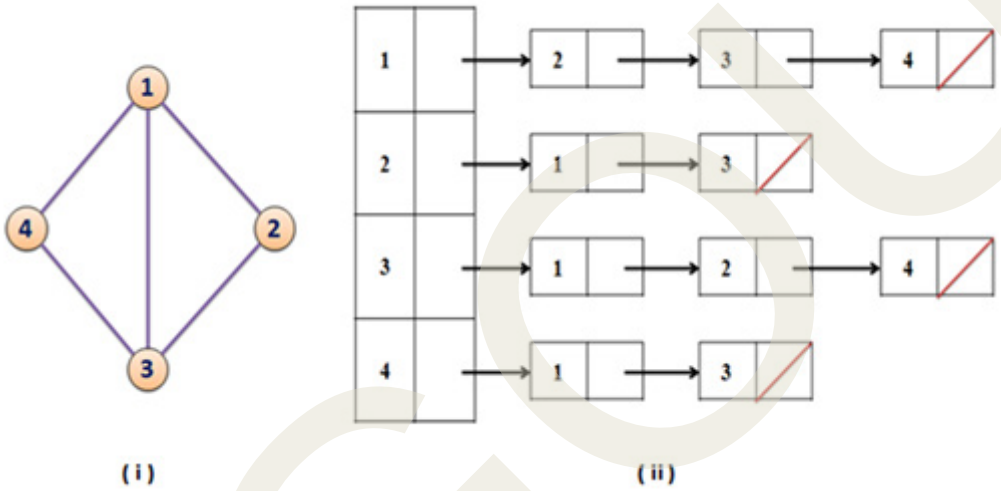


Fig 3.4.31 Adjacency List representation of an Undirected Graph

The next index is 2. That is the vertex 2. Vertex 2 has two adjacent vertices: vertex 1 and vertex 3. So index 2 points to node structure 1, which is the representation of node 1 and then points to node 3 and terminates. In the case of node 3, the adjacent nodes or vertices are node 1, node 2, and node 4. So we can represent the list as node 3 points to 1, then points to 2, then points to 4 and terminates. In the case of node 4, the adjacent vertices are node 1 and node 3. So, the list representation starts from node 4 and points to node 1, then points to node 3 and terminates.

3.4.5.2 Representing a Directed Graph

Consider a digraph shown in Figure 3.4.32(i). The graph has 5 nodes. So, the array indexes include 1, 2, 3, 4, and 5. The linked representation is shown in Figure 3.4.32(ii). We can see 5 separate lists in the adjacency list representation. In the case of node 1, the adjacent nodes are node 2 and node 3. So, in the adjacency list representation, the node 1 points to 2 and then points to 3 and terminates. In the case of node 2, the adjacent node is node 3. So, in the adjacency list representation, the node 2 points to 3 and terminates. In the case of node 3, the adjacent node is node 5. So, in the adjacency list representation, the node 3 points to 5 and terminates.

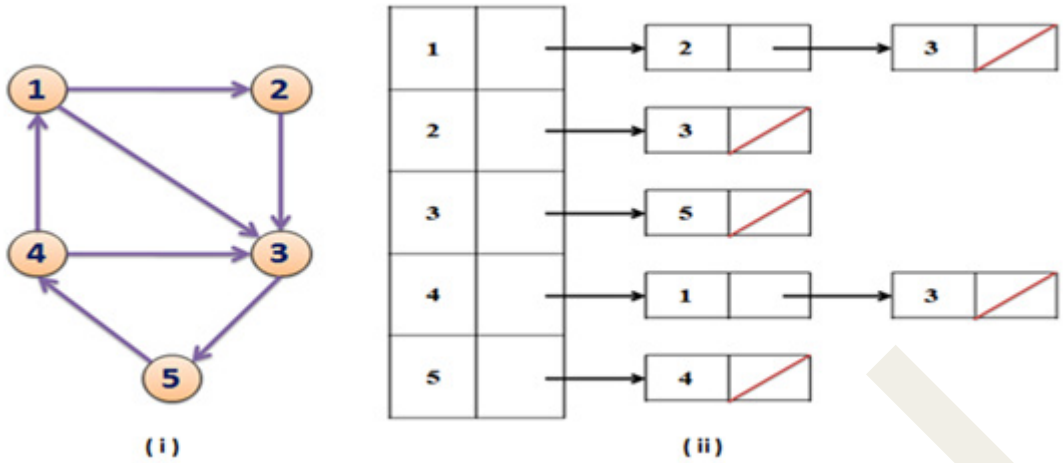


Fig 3.4.32 Adjacency List representation of a Directed Graph

In the case of node 4, the adjacent nodes are node 1 and node 3. So, in the adjacency list representation, the node 4 points to 1 and then points to 3 and terminates. In the case of node 5, the adjacent node is node 4. So, in the adjacency list representation, the node 5 points to 4 and terminates.

3.4.5.3 Multi-List Representation of Graph

Adjacency multi-list representation of a graph is based on the edges. In this graph structure, there are two main parts: a directory of node information and a set of linked lists of edge information. The directory of node information is an array of header nodes. For each node of the graph, there is one entry in the node directory. Each edge in the graph has a separate list representation. The following structure is used to represent the edge information.

Fig 3.4.33 Structure used for storing edge information

- M is a 1 bit field that is used to denote whether or not the node is examined.
- Vertex 1 is the start vertex of an edge (u, v). That is Vertex 1 = u.
- Vertex 2 is the end vertex of an edge (u, v). That is Vertex 2 = v.
- List 1 represents the first down "List name" where Vertex 1 is present.
- List 2 represents the first down "List name" where Vertex 2 is present.

The header nodes are represented in array format. Here, we can call it a directory of node information. The header nodes point to the corresponding edge list.

Consider a simple example to understand the multi-list representation. Figure 3.4.34 shows a simple undirected graph G with four vertices: 1, 2, 3, and 4. There are 6 edges in the graph.

ie. The vertex set is; V(G) = { 1, 2, 3, 4 }

The edge set is; E(G) = {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}

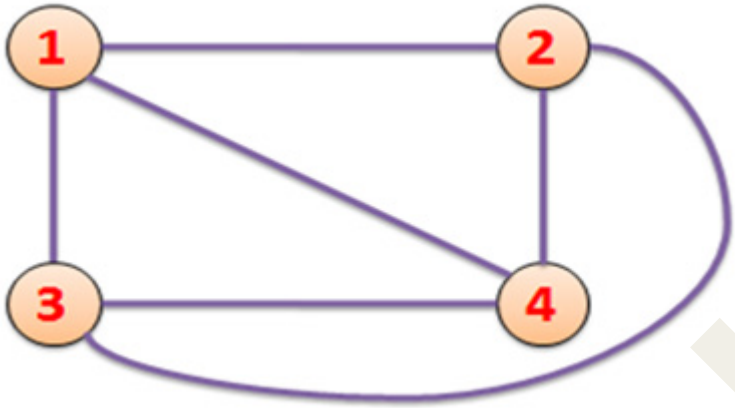


Fig 3.4.34 Example graph for multi list representation

We already know that the adjacency multi list representation is based on edges. The representation contains two parts; the directory of node information, which consists of header nodes, and the edge lists, which include the information about the edges in the given graph. We can see these two parts in Figure 3.4.34.

The header nodes part is represented as an array structure. The nodes 1, 2, 3, and 4 are included in the header nodes section. Each node points to the corresponding edge list. There are 6 edges in this graph. So, 6 edge lists are included in this representation. They are **N1**, **N2**, **N3**, **N4**, **N5**, and **N6**.

- The edge list starts with N1, which represents an edge (1, 2). So, the header node 1 and header node 2 points to N1 because both nodes are connected with this edge. In N1, the start vertex is 1, and the end vertex is 2. The 4th field represents the first down list name where node 1 is present. Here, it is N2 because the first down list, which contains node 1, is N2. (The down list of N1 is N2, N3, N4, N5, and N6.) The 5th field represents the first down list name where node 2 is present. Here it is, N4.

- The second edge list is N2. It represents the edge (1, 3). So, the header node 3 points to N2. In N2, the start vertex is 1, and the end vertex is 3. The 4th field represents the first down list name where node 1 is present. Here, it is N3 because the first down list, which contains node 1, is N3. (Down list of N2 is N3, N4, N5, N6) The 5th field represents the first down list name where node 3 is present. Here it is N4.

- The third edge list is N3. It represents the edge (1, 4). So, the header node 4 points to N3. In N3, the start vertex is 1, and the end vertex is 4. The 4th field represents the first down list name where node 1 is present. Here, no down list contains the node 1. So, we enter a zero or null value to that field. Here, we enter a 0. (Down list of N3 is N4, N5, N6) The 5th field represents the first down list name where node 4 is present. Here it is N5.

- The fourth edge list is N4. It represents the edge (2, 3). In N4, the start vertex is 2, and the end vertex is 3. The 4th field represents the first down list name where node 2 is present. Here, it is N5 because the first down list, which contains node 2 is N5. (Down list of N4 is N5, N6) The 5th field represents the first down list name where node 3 is present. Here it is N6.

- The fifth edge list is N5. It represents the edge (2, 4). In N5, the start vertex is 2, and the end vertex is 4. The 4th field represents the first down list name where node 2 is present. Here, no down list contains the node 2. So, we enter a zero or null value to that field. Here, we enter a 0. (Down list of N5 is N6) The 5th field represents the first down list name where node 4 is present. Here it is N6.

- The sixth edge list is N6. It represents the edge (3, 4). In N6, the start vertex is 3, and the end vertex is 4. The 4th field represents the first down list name where node 3 is present. Here, no down list contains node 3. So, we enter a zero or null value to that field. Here, we enter a 0. (There is no down list for N6) The 5th field represents the first down list name where node 4 is present. Here, no down list contains node 4. So, we enter a zero or null value to that field. Here, we enter a 0.

From this, we can conclude the representation of each vertex by the edge lists as;

Here;

- N1 represents the edge (1, 2).
- N2 represents the edge (1, 3).
- N3 represents the edge (1, 4).
- N4 represents the edge (2, 3).
- N5 represents the edge (2, 4).
- N6 represents the edge (3, 4).

Fig 3.4.35 Multi list representation of example graph

3.4.6 Implementation of Graph

Consider a non weighted graph with the number of vertices MAX, where MAX is

defined as 25.
The basic node type, which stores the information of a non weighted graph, is defined in C as:

```
#define MAX 25        // defining the value of MAX to 25
typedef  struct node       // creating a structure node
 {
   int vertex;       // stores an integer value to the variable vertex
   struct node *next;       // next will point to another structure of the type node
 }node1;   // declaring a structure variable
node1 *adj [MAX];// declaring the adjacency list based on the number of vertices of a graph
```

The above definition is called a structure definition. In this definition, a name is given to the structure. The name of the above structure is node. That is, we are creating a node using structure. This structure has two components. The first is of type int. We declare an integer variable vertex to store the values of vertices or nodes. The second component is a structure of type nodes. Observe that we have defined a structure within a structure. This is allowed in C language. This declaration states that a pointer named next will point to another structure of the type node. That is, one vertex is pointing to the next one.

The use of typedef names the structure as node1. That is, we declare a structure variable node1 for further use of a node. Now, we can declare the adjacency list on the basis of the number of vertices of the graph as node1 *adj [MAX]. That is, we have declared adj[MAX] as a pointer, which will point to data of type node1.

For a weighted graph, the node structure is defined in C language as:

```
#define MAX 25   // defining the value of MAX to 25
typedef  struct node   // creating a structure node
{
  int vertex;   // stores an integer value to the variable vertex
  int weight;    // stores an integer value to the variable weight
  struct node *next;   // next will point to another structure of the type node
}node2;     // declaring a structure variable
node2 *adj [MAX];// declaring the adjacency list based on the number of vertices of a graph
```

The above definition is called a structure definition. In this definition, a name is given to the structure. The name of the above structure is node. That is, we are creating a node using structure. This structure has 3 components. The first and second are of type int. We declare an integer variable vertex for storing the values of vertices or nodes and an integer variable weight for storing the edge weights. The next component is a structure of type nodes. Observe that we have defined a structure within a structure. This is allowed in C language. This declaration states that a pointer named next will point to another structure of the type node. That is, one vertex is pointing to the next one. The use of typedef names the structure as node2. That is, we declare a structure variable node1 for further use of a node. Now, we can declare the adjacency list on the basis of the number of vertices of the graph as node2 *adj [MAX]. That is, we have declared adj[MAX] as a pointer, which will point to data of type node2.

3.4.7 Graph Traversal

Traversal is a searching technique used in graphs. To search for a particular vertex in a given graph, we use traversal. So, the main goal of graph traversal is to find all vertices or nodes that are reachable from a given set of nodes. The traversal also decides the order of vertices visited in a search process. It also finds the edges to be used without creating any loops. That is, in graph traversal, all the vertices are visited without getting into a looping path. We can follow all the edges in an undirected graph. But in a directed graph, we can only follow the out edges.

Traversal in the graph is different from tree traversal because; there is no root node in the graph, so that the traversal can start from any node.

Only those nodes traversed are reachable from the starting node. If we want to traverse all the reachable nodes, we again have to select another starting node for the remaining nodes.

While traversing a graph, there may be a possibility that we will reach a node more than once. To ensure that each node is visited only once, we have to keep the status of each node whether it has been visited or not.

The main graph traversal techniques are Breadth First Search (BFS) and Depth First Search (DFS). We can study each of them in detail in the following sessions. Breadth first search uses a **queue structure** for keeping the nodes and processing. In Depth first search, we use a **stack structure** for node processing.

3.4.7.1 Breadth First Search (BFS)

The BFS technique uses a queue to traverse all the nodes in the given graph. Here, we first take any node as a starting node. That node is placed in the queue at first. So, the front element of the queue is the starting node. Then, we take all the adjacent nodes of the starting node. They are placed behind the starting node in the queue. After entering all the adjacent nodes of the starting node, that node is deleted from the queue. That is, the first node is traversed successfully.

Then, the second node in the queue becomes the front element of the queue. We take

that node for processing. That is, we find all the adjacent nodes and place them in the queue if it is not present in the queue. So, the front node in the queue is traversed successfully, and we delete it from the queue. Then, we take the next front node in the queue. A similar approach is done for all the other adjacent nodes placed in the queue. Likewise, each node is processed.

Algorithm

Step 1. Define a queue for storing the nodes of the graph.

Step 2. Select any node as a starting node for traversal and insert it into the queue.

Step 3. Delete the front element from the queue and insert all its unvisited adjacent nodes into the queue at the end.

Step 4. Repeat step 3 until the queue becomes empty.

Example:

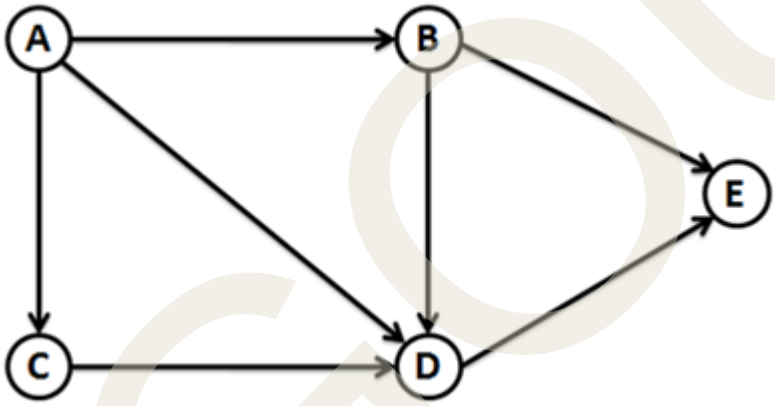


Fig 3.4.36 Example graph for BFS

Let us consider a graph shown in Figure 3.4.36 for breadth first traversing. Take the starting node as node A. The adjacent nodes of A are B, C, and D. The following steps illustrate the BFS.

1. Initially insert the starting node into the queue. That is, insert node A into the queue. Here; Front = Rear = 0. The resultant graph and queue are shown in Figure 3.4.37.

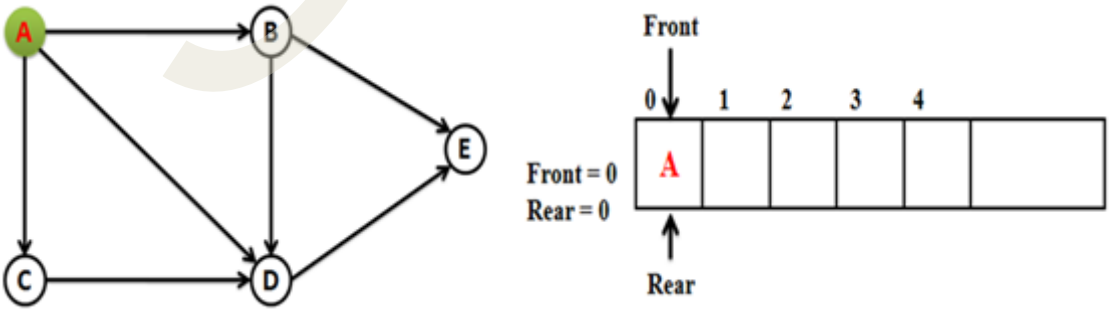


Fig 3.4.37 Breadth First Traversal - Resultant graph, queue (1)

2. Remove the front element A from the queue and increment Front = Front + 1. That is, Front becomes 1. Insert the adjacent nodes of A into the queue if it is not present in the queue. Here, B, C, and D are the adjacent nodes, and they are inserted into the queue.

So the Front = 1 and Rear = 3. Figure 3.4.38 shows the resultant graph and queue.

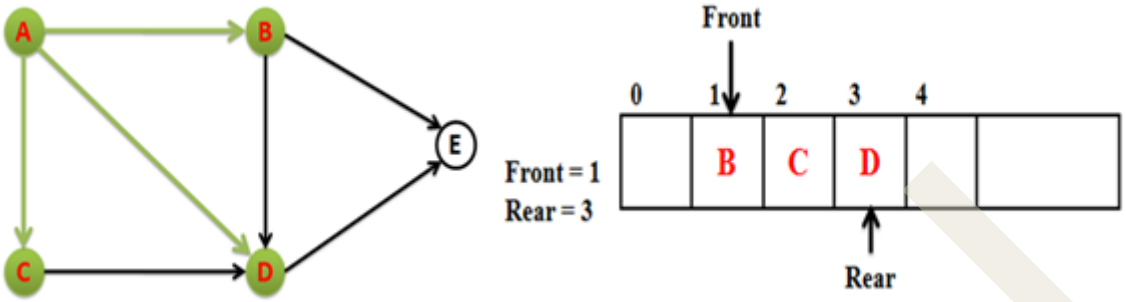


Fig 3.4.38 Breadth First Traversal - Resultant graph, queue (2)

3. Remove the front element B from the queue and add the adjacent nodes of B to the queue if it is not present in the queue. The adjacent nodes of B are D and E. Here, D is already in the queue. E is not in the queue. So insert E to the rear position. So the Rear = 4 and Front = 2. Figure 3.4.39 shows the resultant graph and queue.

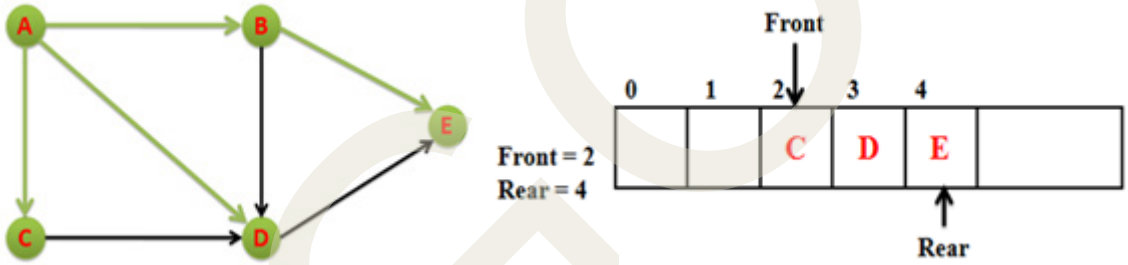


Fig 3.4.39 Breadth First Traversal - Resultant graph, queue (3)

4. Remove the front element C from the queue and add the adjacent nodes of C to the queue if it is not present in the queue. The adjacent node of C is D. Here, D is already in the queue. So the Rear = 4 and Front = 3. Figure 3.4.40 shows the resultant graph and queue.

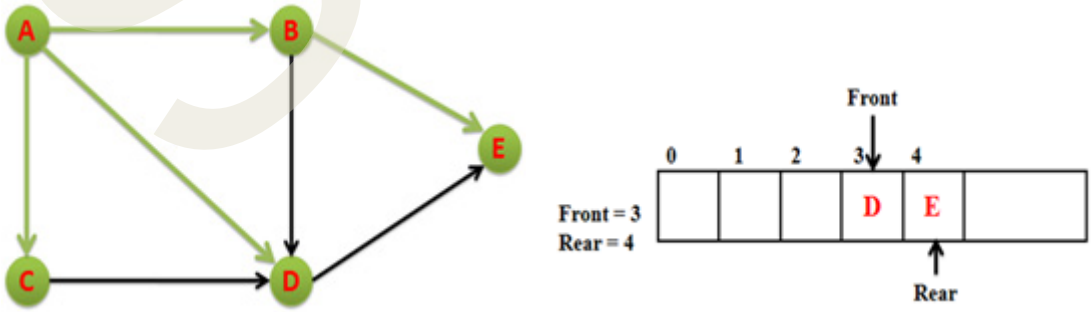


Fig 3.4.40 Breadth First Traversal - Resultant graph, queue (4)

5. Remove the front element D from the queue and add the adjacent nodes of D to the

queue if it is not present in the queue. The adjacent node of D is E. Here, E is already in the queue. So the Rear = 4 and Front = 4. Figure 3.4.41 shows the resultant graph and queue.

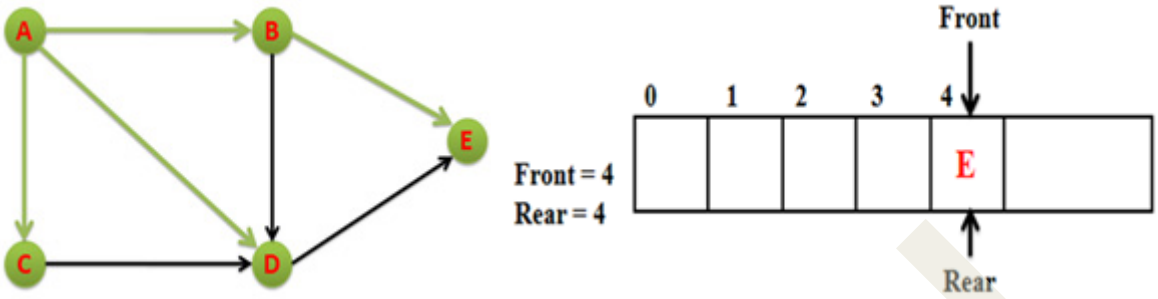


Fig 3.4.41 Breadth First Traversal - Resultant graph, queue (5)

6. The process is repeated until Front & Rear. Remove the front element E from the queue and add the adjacent nodes of E to the queue if they are not present. Here, node E has no adjacent nodes. So the queue becomes empty. Also, Front and Rear. Here, Front = 5 and Rear = 4. So, we stop the traversal. Figure 3.4.42 shows the resultant graph and queue.

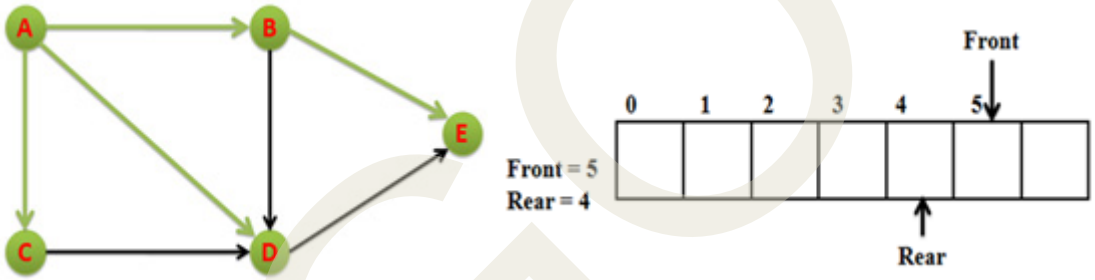


Fig 3.4.42 Breadth First Traversal - Resultant graph, queue (6)

Finally, the traversed nodes are A, B, C, D, E.

3.4.7.2 Depth First Search (DFS)

Depth First Search (DFS) starts from a source node. We can select any node from the graph as a starting or source node. If S1 is the starting node, then DFS first visits S1 and then visits any one of its adjacent nodes, say S2. Then DFS again visits an adjoining node of S2, say S3. In the next step, DFS again visits any one of the adjacent nodes of S3, such as S4 and so on. DFS uses a stack for processing nodes. DFS uses a backtracking method for traversing all unvisited nodes in the graph. For example, if node S4 has no adjacent nodes, then it backtracks the traversal to the previous node. That is, S4 backtracks to S3. Then, DFS looks for any other adjacent node in S3. If it exists, then visit that node. If it does not exist, then backtrack to the previous node S2. This process is repeated until all the unvisited nodes are visited.

Algorithm:

Step 1. Define a stack for storing the nodes of the graph.

Step 2. Select any node as a starting node for traversal and push it into the stack.

Step 3. Visit any one of the unvisited adjacent nodes of a node which is on the top of the stack and push it onto the stack

Step 4. Repeat Step 3 until there is no new node to be visited from the node on the top of the stack.

Step 5. If there is no unvisited adjacent node, it remains to backtrack the traversal and pop the top node from the stack.

Step 6. Again, repeat the Steps 3, 4, and 5 until the stack becomes empty.

Example: Let us consider a graph shown in Figure 3.4.43 for depth-first traversing. Take the starting node as node A.

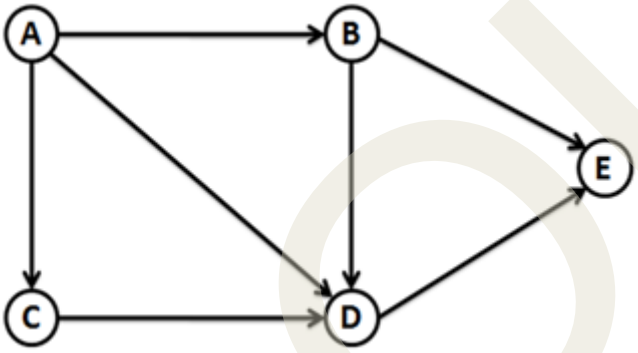


Fig 3.4.43 Example graph for DFS

The following steps illustrate the DFS.

1. Initially insert the starting node into the stack. That is, push node A into the stack. Here, top = 0. Figure 3.4.44 shows the resultant graph and stack.

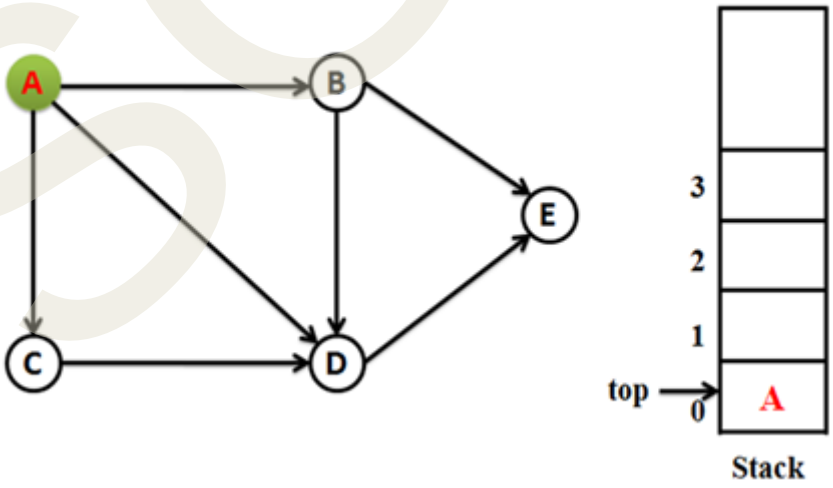


Fig 3.4.44 Depth First Traversal - Resultant graph, stack (1)

2. Visit any one of the unvisited adjacent node of A. Here, we visit node B. Then push B into the stack. So top = 1. Figure 3.4.46 shows the resultant graph and stack.

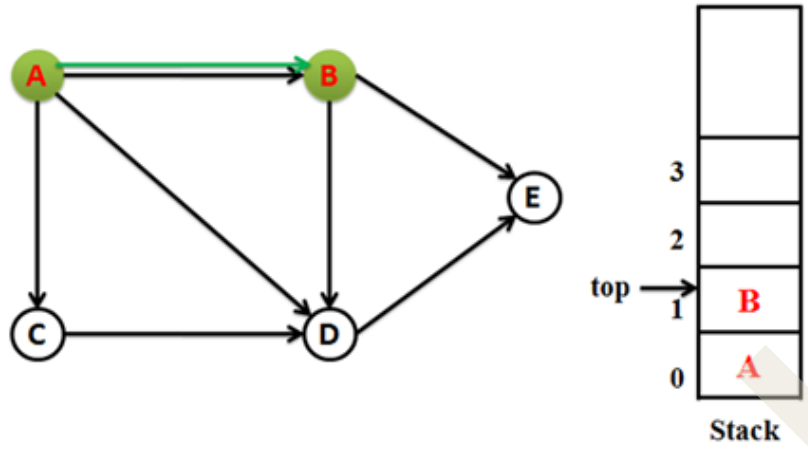


Fig 3.4.45 Depth First Traversal - Resultant graph, stack (2)

3. Visit any one of the unvisited adjacent node of B. Here, we visit node D. Then, push D into the stack. So top = 2. Figure 3.4.46 shows the resultant graph and stack.

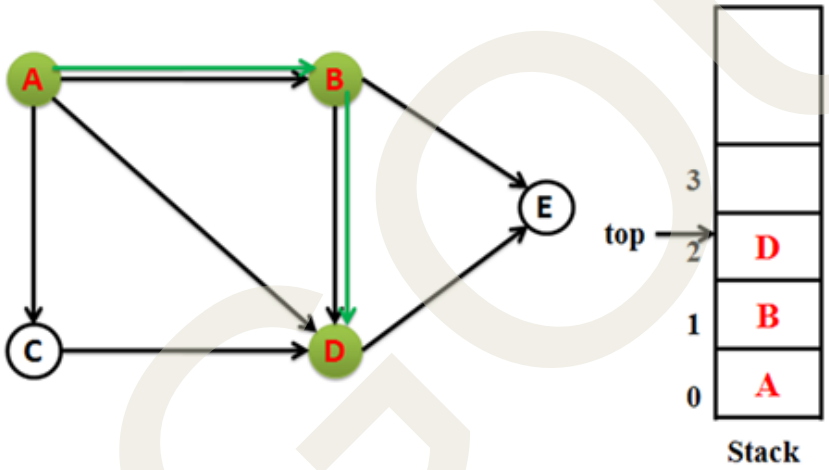


Fig 3.4.46 Depth First Traversal - Resultant graph, stack (3)

4. Visit any one of the unvisited adjacent node of D. Here, the node is E. So we visit the node E. Then push E into the stack. So top = 3. Figure 3.4.47 shows the resultant graph and stack.

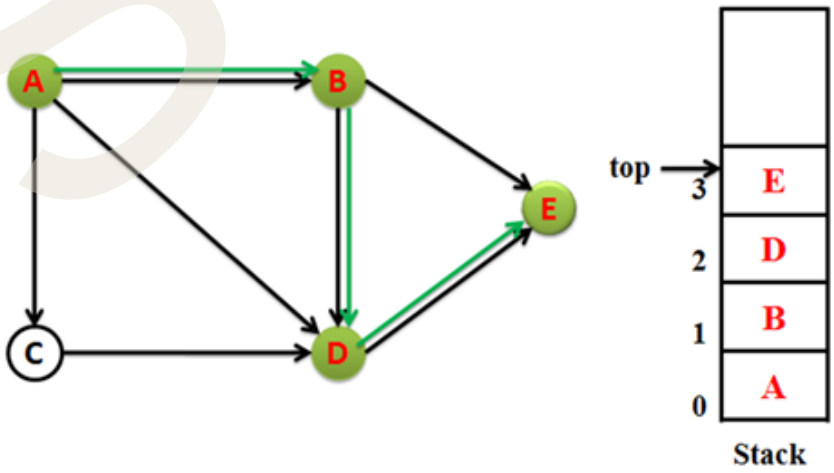


Fig 3.4.47 Depth First Traversal - Resultant graph, stack (4)

5. Visit any one of the unvisited adjacent nodes of E. There are no adjacent nodes for node E. So, we backtrack the traversal. Pop node E from the stack. So top = 2. Figure 3.4.48 shows the resultant graph and stack.

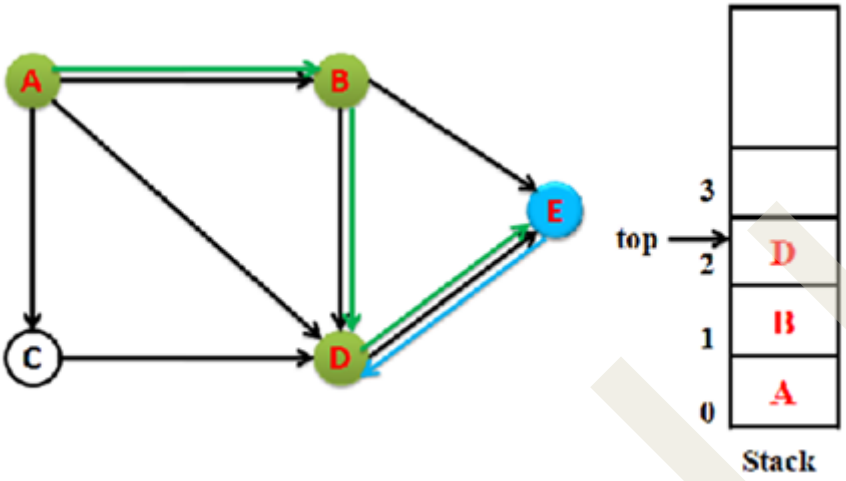


Fig 3.4.48 Depth First Traversal - Resultant graph, stack (5)

6. Visit any one of the unvisited adjacent node of D. There is no other unvisited adjacent node remaining for node D. So we backtrack the traversal. Pop node D from the stack. So top = 1. Figure 3.4.49 shows the resultant graph and stack.

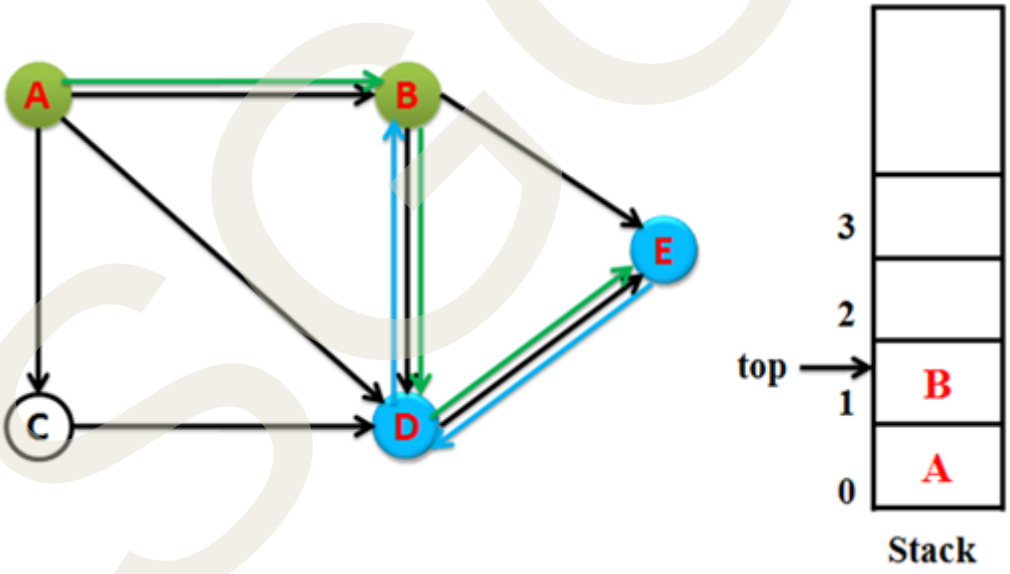


Fig 3.4.49 Depth First Traversal - Resultant graph, stack (6)

7. Visit any one of the unvisited adjacent nodes of B. There is no other unvisited adjacent node remaining for node B. So, we backtrack the traversal. Pop node B from the stack. So top = 0. Figure 3.4.50 shows the resultant graph and stack.

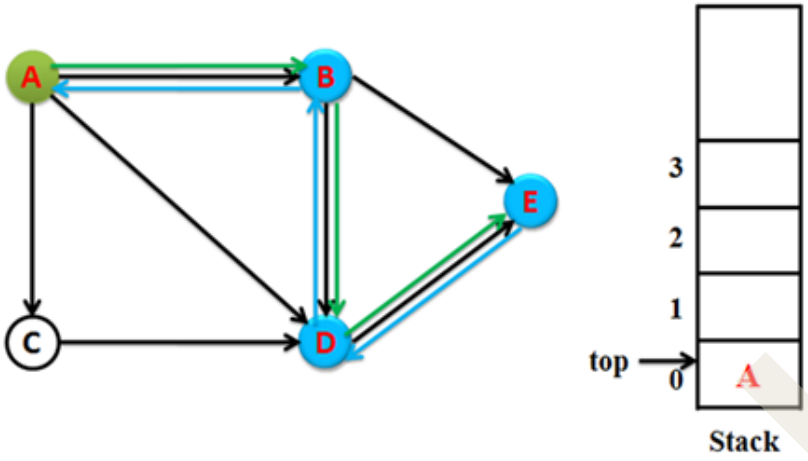


Fig 3.4.50 Depth First Traversal - Resultant graph, stack (7)

8. Visit any one of the unvisited adjacent node of A. Here, only one unvisited node of A remains, and that is the node C. So we visit node C. Then push C into the stack. So top = 1. Figure 3.4.51 shows the resultant graph and stack.

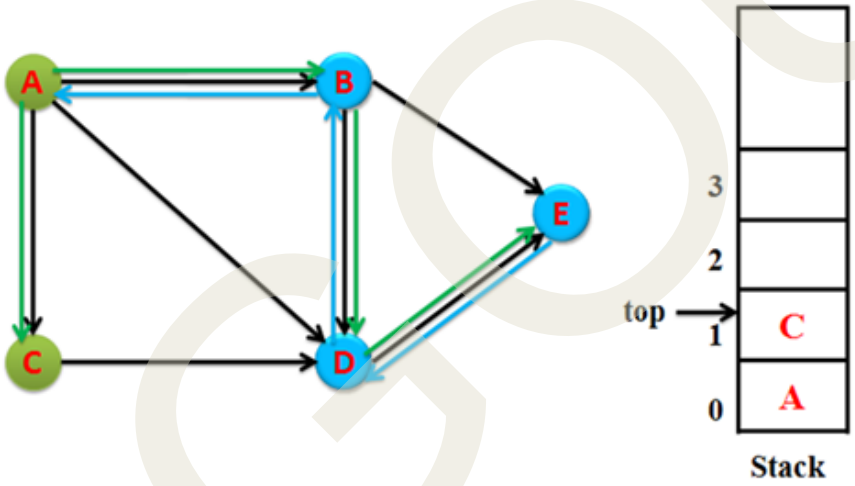


Fig 3.4.51 Depth First Traversal - Resultant graph, stack (8)

9. Visit any one of the unvisited adjacent node of C. There is no other unvisited adjacent node remaining for node C, so we backtrack the traversal. Pop node C from the stack. So top = 0. Figure 3.4.52 shows the resultant graph and stack.

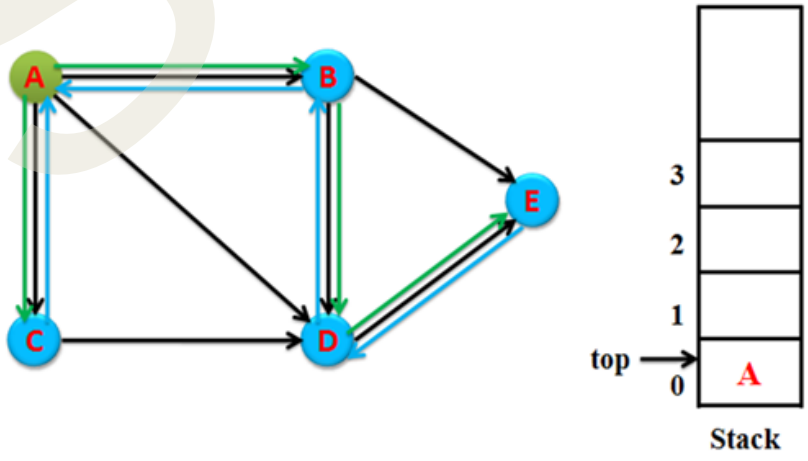


Fig 3.4.52 Depth First Traversal - Resultant graph, stack (9)

10. Visit any one of the unvisited adjacent nodes of A. There is no other unvisited adjacent node remaining for node A. So, we backtrack the traversal. Pop node A from the stack. So top = -1. That is, the stack becomes empty. So we can stop DFS. Figure 3.4.53 shows the resultant graph and stack.

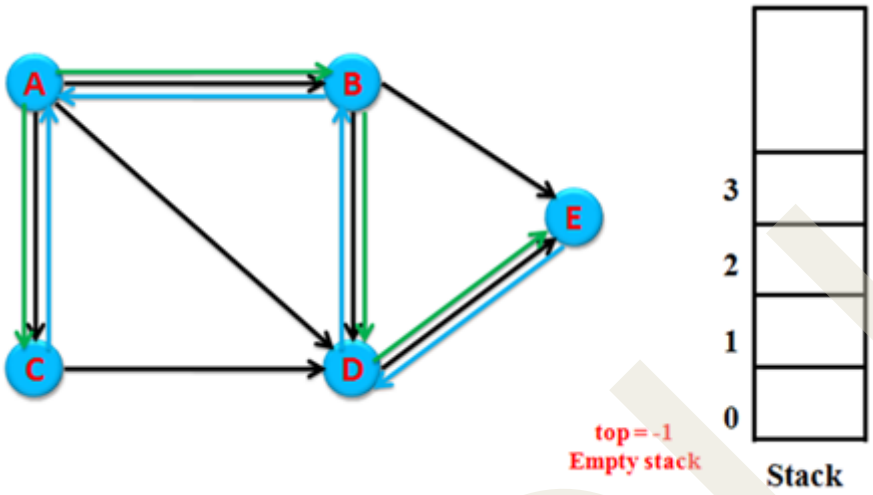


Fig 3.4.53 Depth First Traversal - Resultant graph, stack (10)

And finally, all the nodes are traversed.

Note:

1. Suppose the searching item is very near to the starting vertex, and then BFS is used.
2. If the searching item is far from the starting vertex, then DFS is used.
3. BFS is vertex based traversal.
4. DFS is edge based traversal.

3.4.8 Applications of Graph

Graphs are versatile data structures used to model relationships between entities, making them essential in various applications. Some of the important applications of graphs are travelling salesperson problems, GPS Navigation Systems, Social Networks, Knowledge Graphs, etc.

3.4.8.1 Traveling Salesman Problem (TSP)

Suppose you are an area manager of a financial firm. Your duty is to manage all the branches in your area. Your office is attached to the main branch. But you have to visit all the other branches periodically. Your higher authority insists that you conduct a quick inspection of all branches that are under you and submit the report to him in five days. What will you do first? How will you manage this situation? You have to find the shortest route to visit each branch and return to your office without failure. Proper planning is needed for this purpose. The Traveling Salesman Problem (TSP) is also like this. TSP consists of a salesperson and a set of cities. The salesman has to visit all the cities, starting from a certain one (home town), by selecting the shortest routes and returning to the place where he began. So, the challenge of TSP is to minimise the total length of the trip.

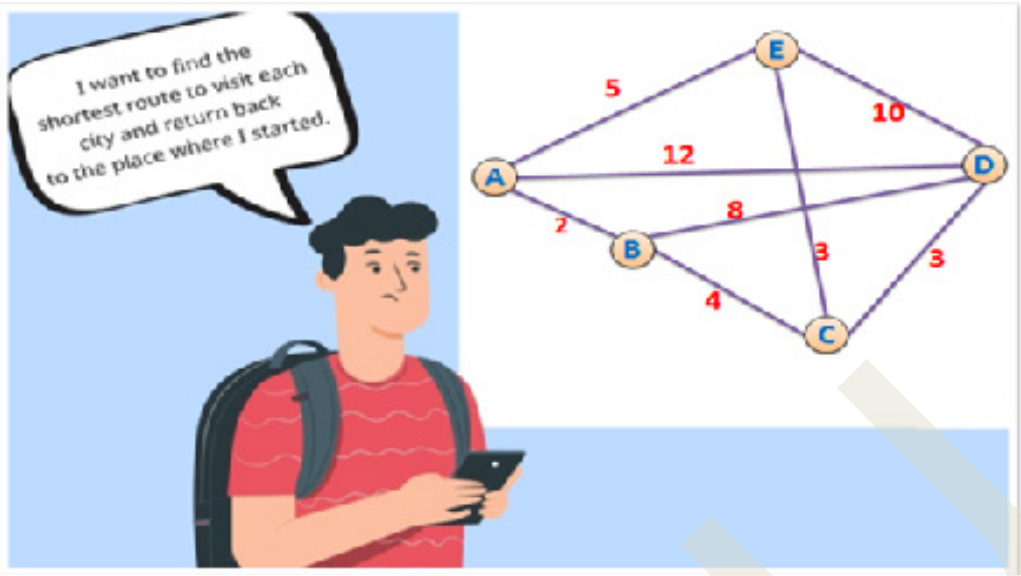


Fig 3.4.54 Traveling Salesman Problem

In Figure 3.4.54, we can see a set of cities represented by a graph. Here, each city is represented by a node. Here, the problem lies in finding a minimal path passing from all nodes at once. Suppose the home town is A. That is, the salesman starts his journey from the city A. He must visit all the cities (B, C, D, and E) and return to his home town (A). So he has to find a route with a minimum distance that starts from A, passes through all the cities once and returns to A. For example, take the path A-B-C-D-E-A. We can represent it as Path 1 = {A, B, C, D, E, A}. And take the path A-B-C-E-D-A. We can represent it as Path 2 = {A, B, C, E, D, A}. Both paths pass all the nodes, but Path 1 has a total length of 24, and Path 2 has a total length of 31. So, considering all other possible paths, the salesman has to choose the shortest route. Here, Path 1 is the shortest route, and it is selected.

3.4.8.2 Google Map

Suppose you are going to attend your friend's wedding in Kunnamkulam. But you have no idea about that place. Now you are staying at Wadakkanchery. What will you do? You will search that place in Google Maps, and you can see (Figure 3.4.55) the different routes to Kunnamkulam from Wadakkanchery. It shows you the distance and approximate travel time to reach the destination.

Google Maps uses graphs to build transportation systems. In Google Maps, various locations are represented as vertices and roads connecting these locations are represented as edges. The intersection of two or more roads is also considered to be a vertex. The navigation system is based on an algorithm that calculates the shortest path from one vertex to another. In the previous example, we can see three paths between Wadakkanchery and Kunnamkulam. We can represent these paths using graphs.

Figure 3.4.55 shows the graph representation. Here, Wadakkanchery is the source vertex represented by vertex 1, and Kunnamkulam is the destination vertex represented by vertex 6. The interconnection of two or more roads that are coming in these different routes (paths) can be represented as vertices (vertices 2, 3, 4 and 5). Here, $\mathbf{w}_{12}$ is the distance between nodes 1 and 2, and $\mathbf{w}_{26}$ is the distance between nodes 2 and 6.

Fig 3.4.55 Graph representation of routes in Google Map

Likewise, w_{13}, w_{35}, w_{56}, w_{24} and w_{46} are the distances between the corresponding nodes. Here, three paths exist between node 1 and node 6. They are; Path 1= 1-2-6, Path 2= 1-2-4-6 and Path 3= 1-3-5-6. Here, the shortest path is path 1. So, it is recommended to the user.

GPS Navigation System

For vehicle navigation, we use the Global Positioning System (GPS). It is also a type of shortest path routing API and differs from Google Maps routing API because it uses a single source (from one vertex to every other). That is, it computes locations from where you are to any other location you might be interested in going. Here, BFS is used to find all the neighboring places. Stand-alone GPS devices actually store their

Fig 3.4.56 Examples of GPS Navigation Systems

map data compared to a smartphone, which is cloud based. It requires a connection to download the maps as you go.

If you don't have a connection, whether it be 3G, 4G or Wifi, you will get a blank screen. Figure 3.4.57 shows examples (MAPMYINDIA, Primo GPS, etc.) for GPS navigation systems.

Flight Networks

For flight networks, an efficient route optimisation is needed. The graph data structure is perfectly fit for this purpose. Using graph models, airport procedures can be modelled and optimised efficiently. Graphs are used to compute shortest paths and fuel usage in route planning. The vertices of in-flight networks are places of departure and destination, airports, aircraft, cargo weights, etc.

The flight trajectories between airports are the edges. Entities such as flights can have properties such as fuel usage crew pairing, which can be more graphs. Figure 3.4.58 shows a simple flight network graph of a particular Airline. It includes airports as vertices and flights between the airports as edges.

From Figure 3.4.58, we can see the Airline operates between 5 five major airports. A particular flight from Cochin to Mumbai is represented as a directed edge from the vertex Cochin to the vertex Mumbai.

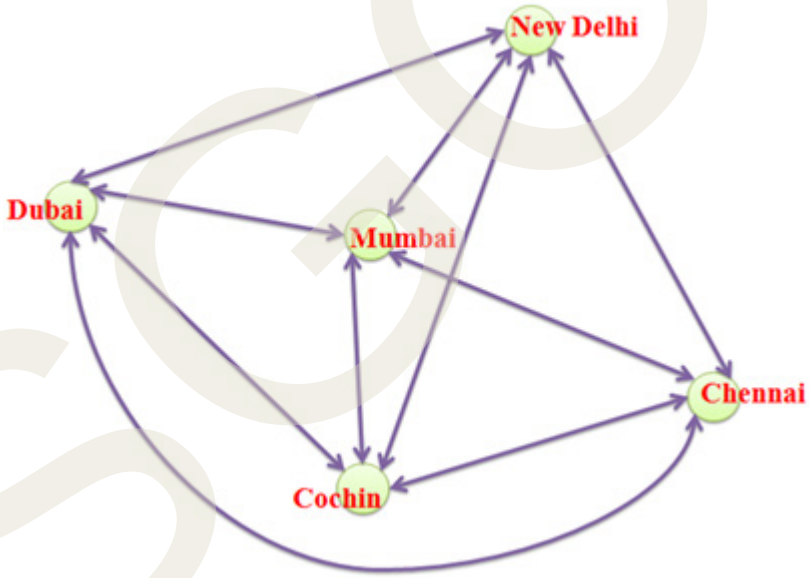


Fig 3.4.57 A simple flight network graph

3.4.8.3 Social Networks

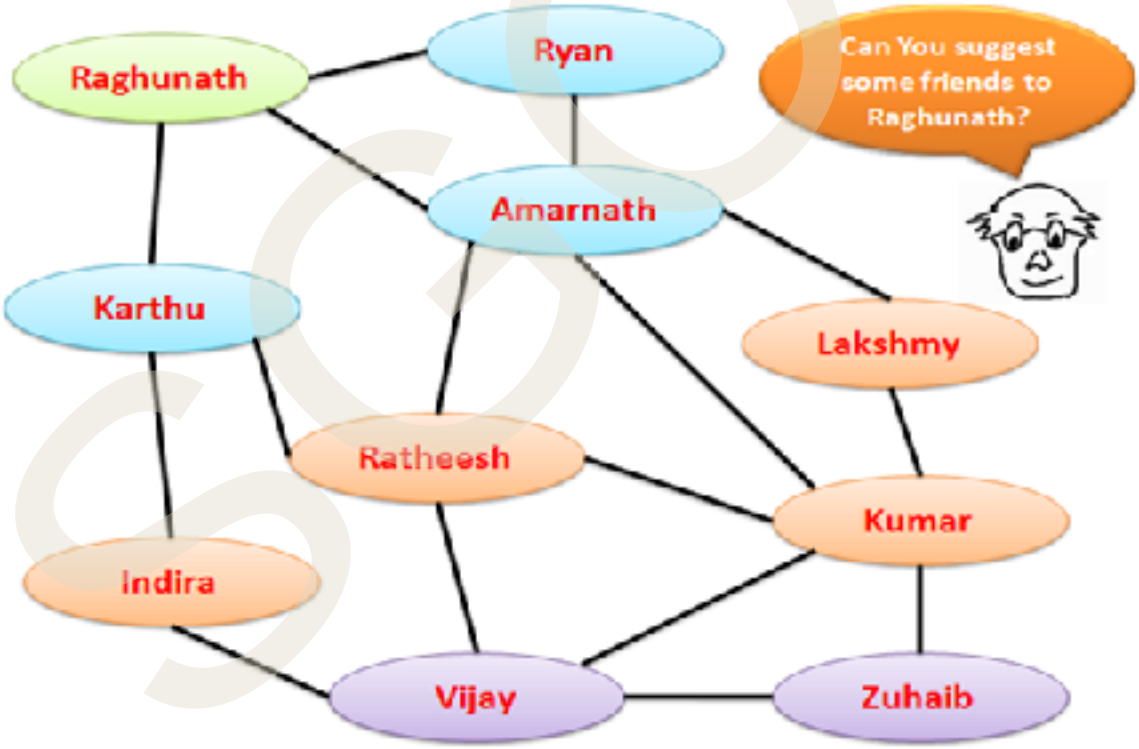
Suppose you are browsing on the Internet, and lots of web pages are loaded into your browser according to your browsing habits. When you click on a link, it will redirect you to some other page. The graph data structure also plays a significant role in the World Wide Web. In WWW, web pages are considered to be the vertices. If a link from page u to page v is there, then there will be an edge from page u to page v.

The best example of the application of graphs in real life is social networks like Facebook and Twitter. Entities like Users, Pages, Places, Groups, Comments, Photos, Photo

Albums, Stories, Videos, Notes, and Events in social networks are represented as vertices or nodes in graphs. Anything that has properties that store data is a vertex. Every connection or relationship is an edge. For example, a user posting a photo, video, comments, etc., are all represented with edges. A user updates his profile with his place of birth, and relationship status is also represented by edges. Suppose you like a photo of your friend; an edge is inserted between you and that photo.

A real social network would have millions and billions of nodes. Figure 3.4.59 shows only a few nodes of a social network. A social network like Facebook can be represented as an undirected graph. Here, each user is represented as a vertex or node. If two users are friends, then there would be an edge between them. Friendship is a mutual relationship. If I am your friend, then you are my friend too. So, connections have to be two-way. That is, Facebook is an undirected graph. In Figure 3.4.59, we can see that a graph represents the friendship between the users. Here, each node is a user.

Now we want to do something like suggest friends to a user. It is a common thing in social networks. Let's say we want to suggest some friends to Raghunath (Figure 3.4.59). One possible approach to do so can be suggesting friends of friends who are not connected already. Raghunath has 3 friends, Amarnath, Ryan and Karthu and the friends of these three that are not connected to Raghunath already can be suggested.



No friend of Ryan is not connected to Raghunath already. Amarnath, however, has 3 friends, Lakshmy, Kumar and Ratheesh, who are not friends with Raghunath. So they can be suggested. Karthu has 2 friends, Ratheesh and Indira, who are not connected to Raghunath. We have counted Ratheesh already. So, we can suggest these 4 users (friends) to Raghunath. We described this problem (suggesting friends to a user) in the context of a social network. This is a standard graph problem. The problem here, in pure graph terms, is finding all nodes having a length of the shortest path from a given

node equal to 2. In this example, the length of the shortest path from Raghunath to Indira, Ratheesh, Kumar and Lakshmy is 2. Twitter is an example of a very large-scale complex graph. Here, users and tweets are represented as vertices. The interactions, such as follows, replies likes, posts and retweets, are represented with edges.

3.4.8.4 Recommendation On E-Commerce Websites

We are all familiar with e-commerce websites like Amazon, Flipkart, etc. Suppose you want to buy an LED TV from one of these e-commerce websites. When you search for a product on this website, you can notice various options about things you want to buy or are interested in. E-commerce websites use a technology called recommendation systems. Recommendation systems track your profile information, like what kinds of products you purchase, which pages you click on, what products you are interested in, etc. Based on your profile data, a recommendation system analyses these data and provides you with recommendations.

So, everyone using these e-commerce websites would receive individual personalised recommendations based on their browsing patterns, purchase history, etc. One way of storing these data is to use a graph data structure.

Figure 3.4.60 Example of User-Product graph

Here, all users and products are stored in nodes on a graph with edges connecting related data. That is, if a user purchases a product, then the user node and the product node are connected with an edge. Consider an example shown in Figure 3.4.60. Here,

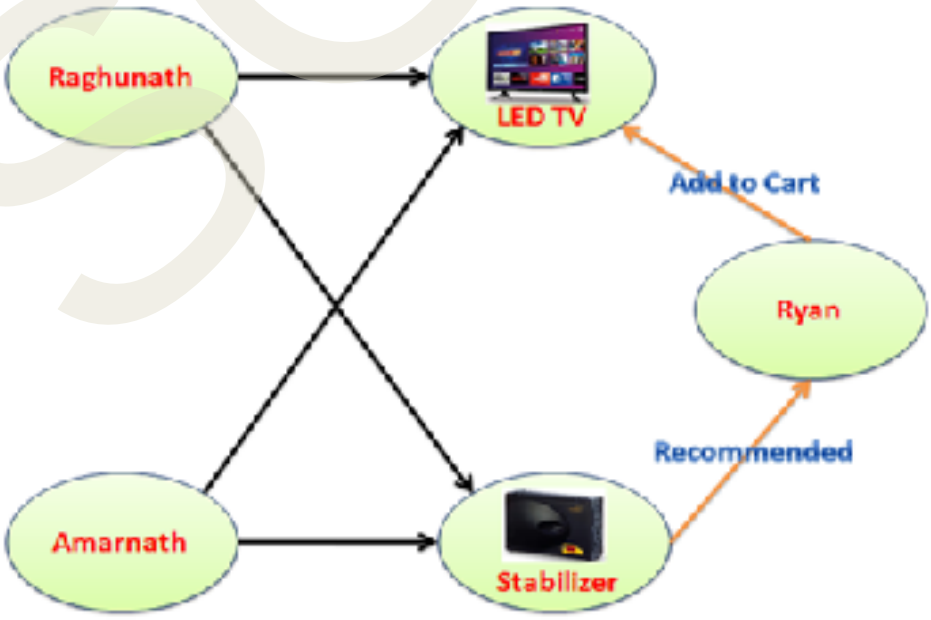


Fig 3.4.61 Example of Recommendation of products using graphs

Raghunath is a user who wants to buy an LED TV. Both the user (Raghunath) and the product (LED TV) are represented by nodes. After purchasing the product, an edge connects both the nodes in the graph representation. Here, the graph shows that Raghunath purchased an LED TV.

We can use the structure of graphs to make recommendations very efficiently. The "Recommendations for you" section on various e-commerce websites uses graph theory to recommend items of similar type to the user's choice. Figure 3.4.61 shows how the recommendation section works. When Ryan adds the LED TV to his cart, the recommendation system looks at all the users connected to the LED TV. In this example, Raghunath and Amarnath are connected to the LED TV. We can find other product nodes that are connected to both Raghunath and Amarnath, such as the Stabilizer, in this example. We then recommend the Stabilizer to Ryan. The graph structure allows you to make these types of requests very quickly.

3.4.9 Knowledge Graphs

Suppose you have a tour plan to visit Paris. You booked your seat with a tour package named Tour Eiffel, which is located in Paris. Your friend is also interested in this package because she wants to see the painting "Mona Lisa" by "Da Vinci". But he didn't get a seat in the Tour Eiffel Package because the booking was closed early. You visited the Louvre Museum and saw the painting "Mona Lisa". After your pleasure trip, you returned to your home. After reaching your house, you want to document your pleasure trip from start to end. You have to convey lots of information from the beginning to the end of the journey. One way is to represent these in a pictorial format. Figure 3.4.62 shows a graphical representation of the context.

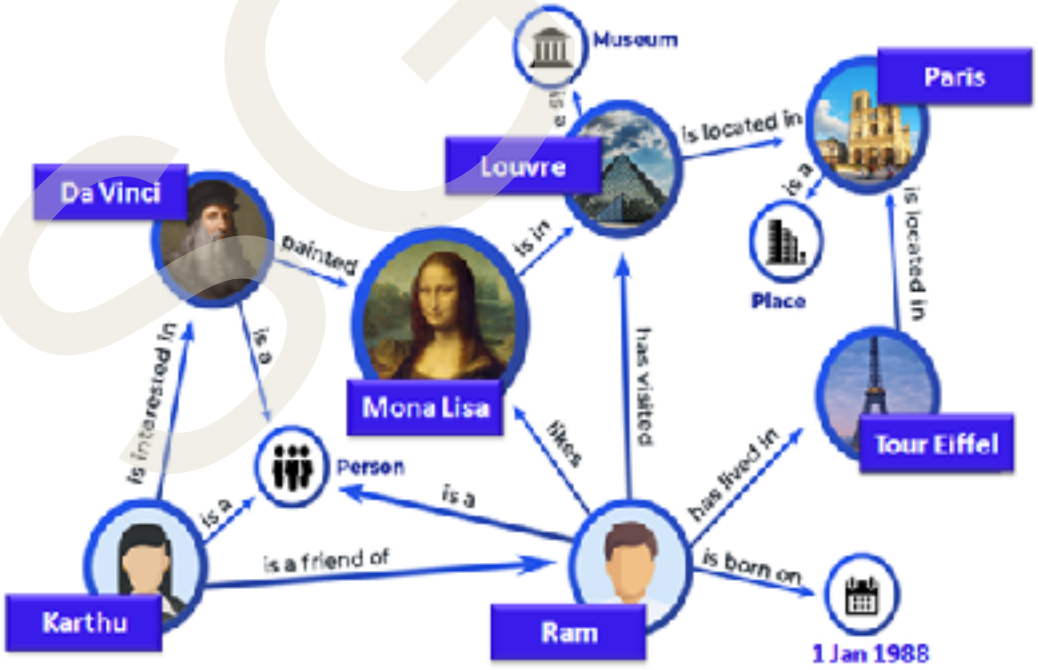


Fig 3.4.62 Example of Knowledge Graph

Here, the entities, objects, events, etc., are represented by nodes, and edges represent the relationship between the nodes. It includes the following information. Karthu (Your friend) is a friend of Ram (You). Both Karthu and Ram are people. Karthu is interested in Da Vinci, and he is also a person. Da Vinci painted the Mona Lisa, which is in the Louvre Museum. Ram likes Mona Lisa. Ram was born on 1st January 1988. Ram has lived in the Tour Eiffel package. It is located in Paris. Paris is a place. Ram has visited the Louvre Museum, which is located in Paris. So, this graph structure integrates data to model a knowledge base. The knowledge base consists of lots of information. This type of representation, shown in Figure 3.4.63, is known as a knowledge graph.

Let's have a look at another interesting example related to the film industry. You may be familiar with Hollywood movies like "Jurassic Park", "Indiana Jones and the Kingdom of the Crystal Skull", and "War of the Worlds" which are directed by the famous Hollywood director Steven Spielberg. All the 3 movies are Sci-Fi movies. Jurassic Park was released on June 11 th, 1993. War of the Worlds was released on June 29 th, 2005 and Indiana Jones and The Kingdom of The Crystal Skull was released on May 22 nd, 2008. We can represent this context using graphs.

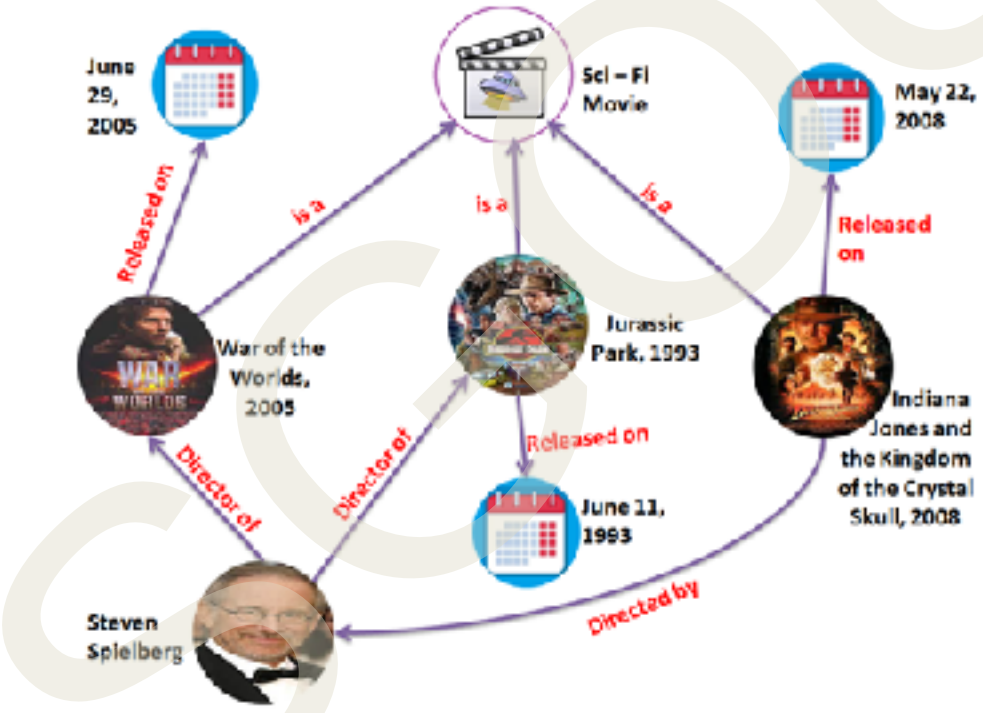


Fig 3.4.63 Knowledge representation using Graphs

Figure 3.4.63 shows the knowledge representation of this context by using graphs. Here, the information is represented with nodes and links. Each link describes the relationship between the nodes. For example, "Steven Spielberg is the director of Jurassic Park." can be defined as;

Here in (1), "Steven Spielberg" and "Jurassic Park" are the two nodes. The relationship between the nodes is "Director of" and is represented on the link. Likewise, all sentences are combined using nodes and links to form a full representation of that par-

ticular context shown in Figure 3.4.63.

Thus, a knowledge graph represents a collection of interlinked descriptions of entities, objects, events or concepts. These entities, objects, events, or concepts are represented as nodes, and the edges represent their relationship. So, a knowledge graph is a knowledge base that uses a graph-structured data model to integrate data.

Recap

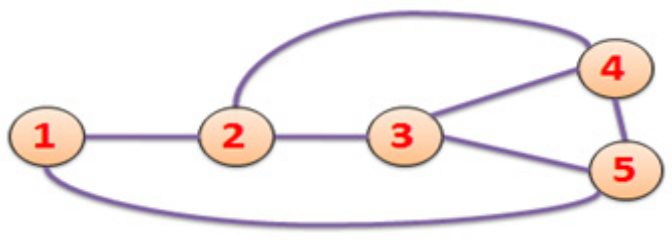
- A graph is a non-linear data structure.
- A graph ***G*** consists of a non-empty finite set of vertices ***V (G)*** and a finite set of edges ***E(G)***.
- A graph can be of two types: Undirected graph and Directed graph.
- If the pair of vertices is unordered, then graph G is called an undirected graph.
- In an undirected graph, if there is an edge between v_1 and v_2, then it can be represented as (v_1, v_2) or (v_2, v_1).
- If the pair of vertices are ordered, then the graph G is called a directed graph or a digraph.
- The line segments or arcs (edges) of the directed graphs have arrowheads which indicate the direction.
- If all the edges in a graph are labelled with some numbers or weights, then the graph is called a weighted graph.
- The degree of a node (vertex) in an undirected graph is the number of edges connected (incident) to that node.
- In a digraph, the degree of a vertex is the number of edges coming to that vertex or edges incident to it, and the out degree of a vertex is the number of edges going outside from that node or the edges incident from it.
- A graph or digraph that does not only have self-loop or parallel edges is called a simple graph.
- A graph which has either a self-loop or parallel edges or both is called a multi-graph
- If n is the number of vertices of a simple undirected graph, then the maximum number of edges can be n(n-1)/2.
- If n is the number of vertices of a simple directed graph, then the maximum number of edges can be n(n-1).
- A graph is said to be complete if each vertex is adjacent to every other vertex in the graph.
- A graph is regular if every node is adjacent to the same number of nodes.
- A graph is planar if it can be drawn in a plane without any two edges intersecting.
- In graph ***G***., a **walk** is a finite sequence of edges of the form v_0v_1, v_1v_2, v_2v_3, ……., $v_{n-1}v_n$.

- A walk in which all the edges and vertices are distinct is called a path.
- If there is a path containing one or more edges that start from a vertex and terminate at the same vertex, then the path is called a cycle.
- A graph that has cycles is called a cyclic graph.
- If a graph does not have any cycle, then it is called an acyclic graph.
- A graph is said to be connected if there is a path from any node of the graph to any other node.
- If, on removing a node from the graph, the graph becomes disconnected, then that node is called the articulation point.
- If, upon removing an edge from the graph, the graph becomes disconnected, then that edge is called the bridge.
- A graph with no articulation points is called a biconnected graph.
- In adjacency list representation, the number of lists depends on the number of vertices. Thus, the adjacency list represents a graph as an array of linked lists.
- Linked representation is a space-saving method that allows the storage of parallel edges.
- Adjacency Multi-lists are edge-based graph representations.
- The main parts of adjacency multi list representation are a directory of node information and a set of linked lists of edge information.
- The directory of node information is an array of header nodes.
- For each node of the graph, there is one entry in the node directory and the header nodes point to the corresponding edge list.
- Each edge in the graph has a separate list representation.
- To represent the edge information, a 5 field structure is used.
- Graph Traversal is a searching technique used in graphs.
- There is no root node in the graph, so that the traversal can start from any node
- In graph traversal, all the vertices are visited without getting into a looping path.
- The BFS technique uses a queue for traversing all the nodes in the given graph.
- DFS uses a stack to traverse all the nodes in the given graph.

- In TSP, the salesman has to visit all the cities starting from their home town by selecting the shortest routes and returning to the place where he began.
- Google Maps uses graphs to build transportation systems.
- In Google Maps, various locations are represented as vertices and roads connecting these locations are represented as edges.
- For vehicle navigation, a Global Positioning System or GPS is used.
- The vertices of in-flight networks are places of departure and destination, airports, aircraft, cargo weights, etc. The flight trajectories between airports are the edges.
- In the World Wide Web (WWW), web pages are considered to be the vertices. If a link from page ***u*** to page ***v*** is there, then there will be an edge from page ***u*** to page ***v***.
- The entities like Users, Groups, Comments, Photos, Stories, Videos, Notes, Events, etc., in social networks are represented as vertices or nodes in graphs.
- A real social network would have millions and billions of nodes.
- E-commerce websites use a technology called recommendation systems.
- Recommendation systems track information like what kinds of products you buy, which pages you click on, what products you are interested in, etc.
- Based on your profile data, a recommendation system analyses these data and provides you with recommendations.
- The knowledge graph represents a collection of interlinked descriptions of entities, objects, events or concepts.

Objective Type Questions

1. A graph is non-linear in structure, which is made up of vertices that are connected by edges. True or False?
2. Write down the number of vertices and the number of edges in the following graph.



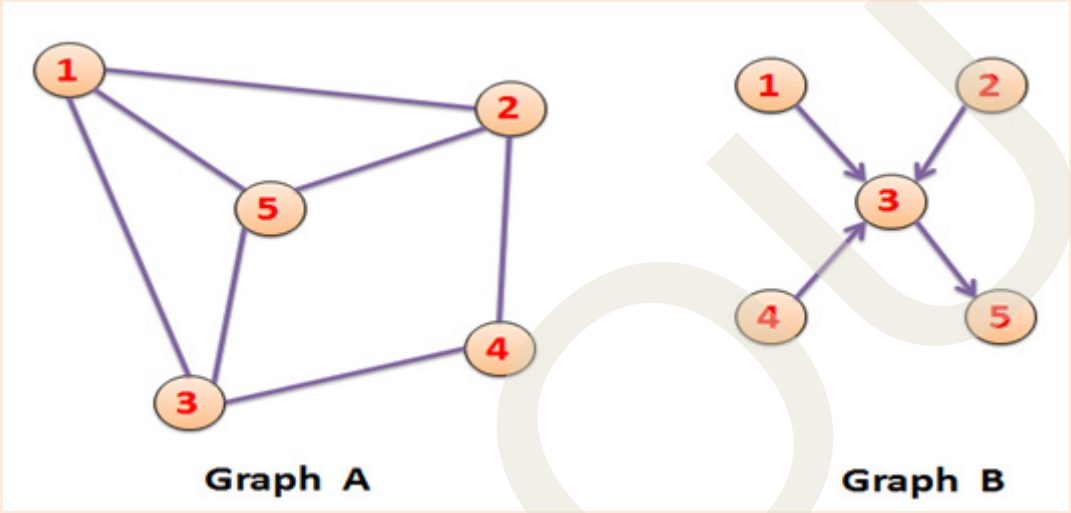
3. Write examples for non-linear data structures.

4. The set of edges contains _____.

5. What are the two types of graphs?

6. What is the name of graph G If the pair of vertices is unordered?

7. From the graphs shown below, answer the following questions:



(i) Graph A is a directed graph. True or False?

(ii) Graph B is a digraph. True or False?

(iii) What is the name of Graph A?

(iv) What is the name of Graph B?

(v) The edge set of graph A contains (5, 3). True or False?

(vi) The edge set of graph B contains (5, 3). True or False?

8. What is the name of graph G If the pair of vertices is ordered?

9. For graph G, if there is more than one edge between the same pair of vertices, what is the name of that graph?

10. In which type of graph, if (v 1, v 2) is an edge, then v2 is adjacent to v1?

11. What is the term used to represent the number of edges connected to a node in an undirected graph?

12. The indegree of a vertex in a digraph is the number of edges coming to that vertex or edges incident to it. True or False?

13. What is the name of the graph that does not have only self-loop or parallel edges?

14. What is the name of a graph which has either a self-loop or parallel edges or both?

15. For a complete graph G, how is each vertex related to every other vertex?

16. In a directed graph, the edge (u, v) is incident from node u, and is incident to node v true or false?

17. For a graph G, if there is an edge whose starting and ending vertices are the same, then it is called a self loop. True or False?

18. In which type of graph is every node adjacent to the same number of nodes?

19. Which type of graph can be drawn in a plane without any two edges intersecting?

20. What can be found on every pair of distinct vertices in a connected graph?

21. What is the term used to represent a walk in a graph in which all edges are distinct?

22. All trails are paths. True or False?

23. Any loop or pair of multiple edges is a cycle. True or False?

24. What is the term used to represent a graph that has cycles?

25. What is the term used to represent a graph that does not have any cycle?

26. If a bridge is removed from a graph, then the graph becomes disconnected. True or False?

27. What is the term used to represent a graph with no articulation points?

28. Which matrix keeps the information of adjacent nodes?

29. In an undirected graph G, an edge exists from node u to node v. What is the value of vu?

30. What are the entries of the bit matrix?

31. The adjacency matrix of a directed graph is always symmetric. True or False?

32. What is the degree of any vertex 'i' in the adjacency matrix of an undirected graph?

33. The entries of the adjacency matrix of a weighted graph are the weights of the edges between the vertices. True or False?

34. What is the space needed to represent a graph using its adjacency matrix?

35. What is the out degree of any vertex i of the adjacency matrix of a directed graph?

36. What is the degree of any vertex i of the adjacency matrix of a directed graph?

37. What is the other name for the adjacency list representation?

38. An adjacency list is an array of separate lists. True or False?

39. In adjacency list representation, the number of lists depends on the number of edges. True or false?

40. A directed graph has 7 vertices. What is the total number of lists included in the linked representation?

41. Which representation of the graph allows storing parallel edges?

42. The degree of vertex v in a graph G is 3. How many elements are in the adjacency list corresponding to vertex v?

43. Which type of graph representation is adjacency Multi-lists?

44. In adjacency multi list representation, what are the main two parts?

45. What is the directory of node information?

46. For each node of the graph, there is one entry in the node directory. True or False?

47. What is the main goal of graph traversal through a given set of nodes?

48. The traversal decides the order of vertices visited in a search process. True or False?

50. In graph traversal, all the vertices are visited without getting into a looping path. True or False?

51. The graph traversal can start from any node. True or False?

52. What structure is used by Breadth first search for keeping the nodes and processing?

53. In BFS, Which node is selected as a starting node for traversal and inserts it into the queue?

54. What will be done in BFS after traversing a node successfully from the queue?

55. What is the condition that the deletion process will conclude?

56. All the edges in a digraph are traversed in BFS. True or False?

57. What type of technique is Depth first search?

58. What is used by DFS for processing nodes?

59. Which node is selected as a starting node in DFS?

60. Which technique is used by DFS for traversing all unvisited nodes in the graph?

61. While performing DFS, no looping paths are created. True or False?

62. DFS visits any one of the unvisited adjacent nodes of a node which is on the top of the stack and pushes it onto the stack. True or False?

63. If there is a new node to be visited from the node on top of the stack, DFS backtracks the traversal. True or False?

64. In DFS, the edge information of a graph is stored in the stack. True or False?

65. To find the shortest path between two vertices in a graph, we have to see all the possible paths that exist between the vertices. True or False?

66. The challenge of TSP is to minimise the total length of the trip. True or False?

67. How many locations have to be visited in TSP by the salesman after starting from a home town?

68. What is used in Google Maps to build transportation systems?

69. The navigation system used in Google Maps is based on the algorithm to calculate the shortest path from one vertex to another. True or False?

70. In Google Maps, what are represented as vertices?

71. In Google Maps, what are represented as edges?

72. Global Positioning System or GPS is used for vehicle navigation. True or False?

73. There is no difference between Google Maps and GPS routing APIs'. True or False?

74. The vertices in-flight networks are places of departure and destination, airports, aircraft, cargo weights, etc. True or False?

75. What is to form edges in-flight networks?

76. Write an example of the GPS navigation system used in India.

77. A real social network would have millions and billions of nodes. True or False?

78. What is the name given to anything that has properties that store data in social networks?

79. What is the term used to represent every connection or relationship in a social network?

80. Nodes represent a user posting a photo, video, or comment. True or False?

81. Facebook is an example of which type of graph?

82. Write an example of a very large-scale complex graph.

83. Which technology is used by e-commerce websites to provide recommendations to the user?

84. A knowledge graph is a knowledge base that uses a graph-structured data model to integrate data. True or False?

85. What is used to represent the relationship between Entities, objects, events or concepts?

Answers to Objective Type Questions

1. True.
2. No. of vertices = 5, No.of edges = 7
3. Tree, graph
4. Pair of vertices
5. Undirected graph and Directed graph.

6. An undirected graph.

7. (i) False (ii) True (iii) an undirected (iv) a directed (v) True (vi) False

8. A directed graph or a digraph.

9. parallel edges.

10. Undirected graph.

11. The degree of the node.

12. True

13. A simple graph.

14. A multi graph.

15. Adjacent

16. True.

17. True

18. Regular

19. Planar

20. Path

21. A trail

22. False

23. True

24. Cyclic graph.

25. Acyclic graph.

26. True.

27. Biconnected

28. Adjacency matrix.

29. 1

30. Either 0 or 1

31. False

32. Row sum

33. True

34. n*n bits

35. Row sum.

36. Column sum.

37. Linked representation.

38. True

39. False

40. 7

41. Adjacency list

42. 3

43. An edge based

44. Vertex List and Edge Lisr.

45. Vertex List.

46. True

47. All nodes reachable

48. True

49. False

50. True

51. Queue

52. Any node

53. Deletes that node from the queue

54. The queue becomes empty

55. False

56. Graph traversal

57. Stack

58. Any node

59. Backtracking

60. False

61. True

62. False

63. True

64. True

65. True

66. all the cities

67. Graphs

68. True

69. Various locations

70. Roads connecting these locations

71. True

72. False

73. True

74. Flight trajectories between airports

75. MAPMYINDIA, Primo GPS

76. True

77. A vertex

78. An edge

79. True

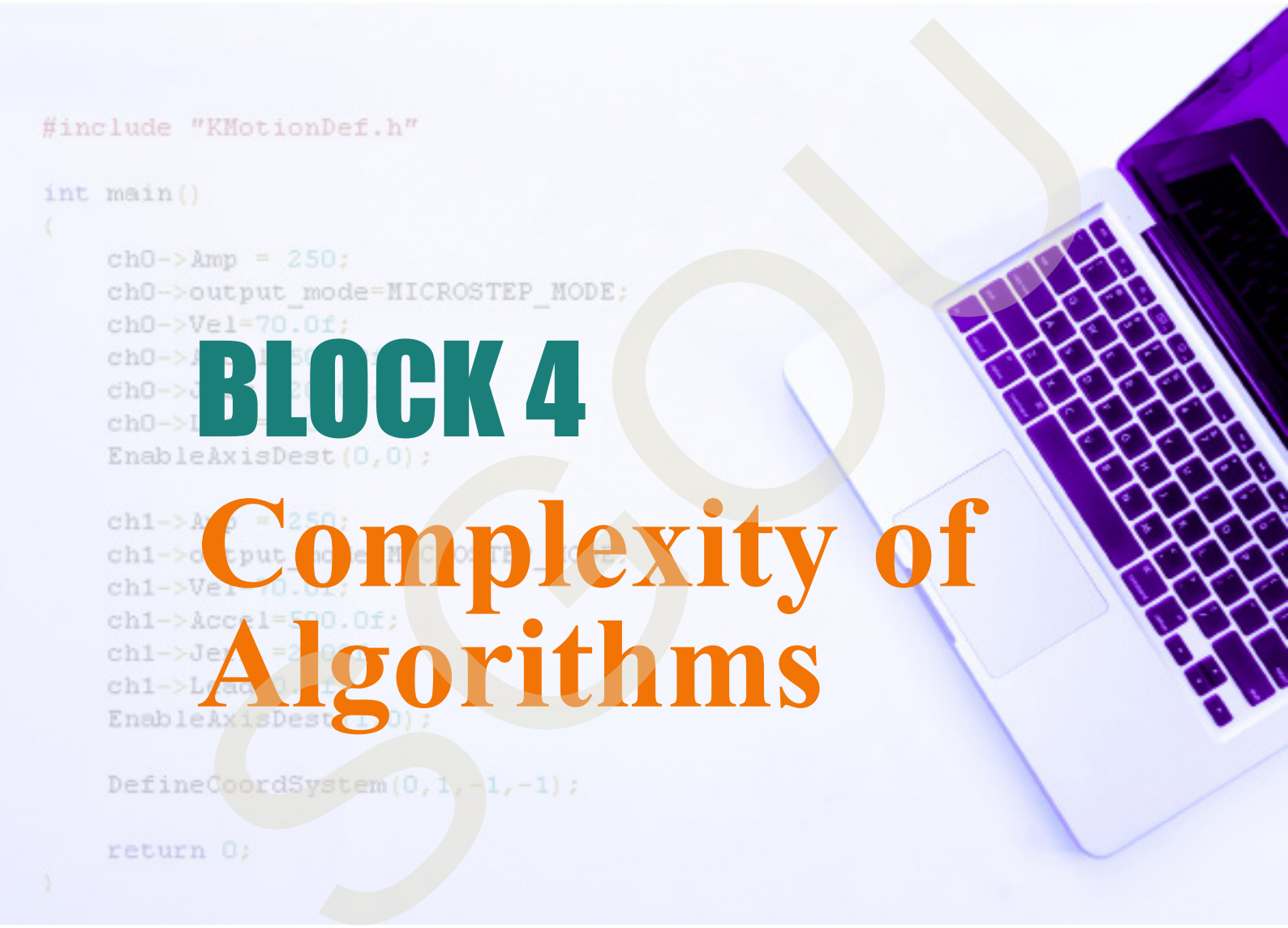
80. Facebook can be represented as both a directed and undirected graph depending on the type of relationships or interactions being modelled.

81. Twitter

82. Recommendation systems

83. True

84. The edges



BLOCK 4

Complexity of Algorithms

Complexity of Algorithms

Learning Outcomes

By completion of this unit, the learner will be able to:

- explain the properties of an algorithm
- introduce different complexities of algorithms according to the input
- make the student aware of polish notations
- familiarise different asymptotic notations in algorithm designing

Prerequisites

Let us consider the problem of preparing a pizza. To prepare a pizza, we follow the steps given below:

1) Get a bowl.

2) Get the white flour.

a. Do we have flour?

i. If yes, put it in the bowl.

ii. If not, do we want to buy flour?

1) If yes, then go out and buy.

2) If not, we can terminate.

3) Turn on the oven, etc…

What we are doing is providing a step-by-step procedure for solving a given problem (preparing a pizza). The formal definition of an algorithm can be stated as follows: An algorithm is a step-by-step unambiguous instruction to solve a given problem. There might be different methods of preparing Pizza. In the same manner, there are other possible algorithms.

However, we have to analyse which one is the best one that will save time and produce the best pizza. In the same manner, algorithms are examined to see which one saves time and space.

Key Concepts

Time complexity, Space complexity, Asymptotic notations, Best case, Worst case, Average case

Discussion

4.1.1 Essential Properties of Algorithms

An algorithm is a finite set of instructions that accomplishes a particular task. The following are the properties of an algorithm:

1. **Input:** An algorithm has zero or more inputs. These are the data items that are given to the algorithm initially before it starts executing.

2. **Output:** After executing an algorithm, we must get at least one output. This output is the result of the algorithm's processing of the input.

3. **Definiteness:** Each step of the instructions must be unambiguous. This property of an instruction is known as definiteness.

4. **Finiteness:** Finiteness refers to the property that the algorithm terminates after a finite number of steps.

5. **Effectiveness:** Effectiveness refers to the property that every instruction must be feasible and that each instruction should do some tasks.

4.1.2 Cases to Consider During Analysis

Let us take our pizza-baking example again. Let us consider your mother and your friend's mother, who are going to prepare pizza. Suppose your mother is very systematic and keeps every ingredient ready before preparing the pizza. In the case of your mother, the performance will be the best if we have all the ingredients ready and everything well arranged, as it will be easy to prepare. Let us imagine that your friend's mother is not very systematic. In this case, even if we have all the ingredients ready and everything well arranged, she might not be able to perform well. However, there might be a cooking style that she is used to, and she can prepare well. This situation might not be favourable for your mother. i.e., the preparation is highly dependent on the kind of input (here, ingredients) that you have provided.

In the same manner, the performance of an algorithm has a significant impact based on the inputs considered when analysing an algorithm. For example, if an input list is almost in the sorted order, some sorting algorithms will perform well, while some other sorting algorithms will perform poorly. However, the opposite would be the condition if the list is randomly arranged instead of sorted. Hence, while analysing an algorithm, multiple input sets must be considered.

The following are the cases to be considered during the analysis:

1. Best Case Input
2. Worst Case Input
3. Average Case Input

4.1.2.1 Best Case Input

Let us take the example of your mother preparing the pizza where all the ingredients are ready and kept in an orderly fashion. Suppose she wanted to search for salt, and since they are kept in an orderly fashion, she would get it easily. In this case, she will take the shortest amount of time to prepare.

Now, let us consider an example of an algorithm for searching a number from a group of numbers. If the number to be searched is found as the first number in the list, then the least amount of time is taken for the search. Such an input is known as best-case input. i.e. best case input represents the input set that allows the algorithm to perform quickest. This is because input takes the shortest time to execute, as it causes the algorithm to do the least amount of work.

4.1.2.2 Worst case Input

Let us take the example of your mother preparing the pizza where all the ingredients are not ready and are not kept in an orderly fashion. Suppose she wanted to search for salt; since they are not kept in an orderly fashion, she would not get it easily, and she would have to search the entire kitchen for the same. In this case, she will take the maximum amount of time to prepare.

Now, let us again consider the example of an algorithm for searching a number from a group of numbers. If the number to be searched is found as the last number in the list, then the maximum amount of time is taken for the search. Such an input is known as worst-case input. i.e., worst-case input represents the input set that allows the algorithm to perform at its slowest. This is because input takes maximum time to execute, as it causes the algorithm to do the maximum amount of work.

4.1.2.3 Average Case Input

Let us take the example of your mother preparing the pizza where some of the ingredients are available and some are not. In this case, she will take some more time than compared to her actual speed as she might have to ask you to purchase some ingredients, arrange ingredients, whatever is available, etc. In this case, she will take an average amount of time to prepare.

In the same manner, average case input represents the input set that allows an algorithm to deliver an average performance.

4.1.3 Complexity of Algorithms

There must be some criteria to measure the efficiency of the algorithm. Measuring the efficiency of algorithms helps compare them. The time and space used by the algorithm are the two main measures of efficiency.

4.1.3.1 Time complexity

The time required to complete a task is crucial. For instance, in cooking, if all the ingredients are prepared in advance and the mother uses a simple cooking method, the time needed will be minimal. However, if she doesn't organise the ingredients and follows a more complex method, the cooking time will be significantly longer.

Time complexity is the amount of time taken by an algorithm to run as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.

4.1.3.2 Space complexity

The correct amount of ingredients in cooking is crucial. Similarly, in our algorithm, the memory usage by each variable is very important.

The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of the characteristics of the input. It is the memory needed for an algorithm until it executes completely.

4.1.4 Estimating complexity

4.1.4.1 Time for an algorithm to run T(n)

Let us consider a situation where you have some numbers that are not in a sorted order. Let us consider the following numbers in random order.

9	4	6	2	5	3

Various techniques were used to sort the above list. Before we consider the cases, let us define the numbers in terms of n.

9	4	6	2	5	3
n1	n2	n3	n4	n5	n6

Let us consider the following case:

Case 1:

In Insertion sort, you compare the key element with the previous elements. If the previous elements are greater than the key element, then you move the previous element

to the next position. Let us make a simple illustration to understand this example:

[9 4 6 2 5 3]

Step 1:

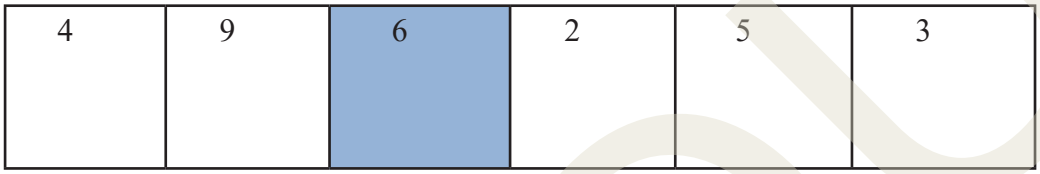
Key =4

The key is compared with the previous element. i.e. the key is compared with 9.

Since 9>4, move the element 9 to the next position and insert 'key' to the previous position.

Result: **[4 9 6 2 5 3]**

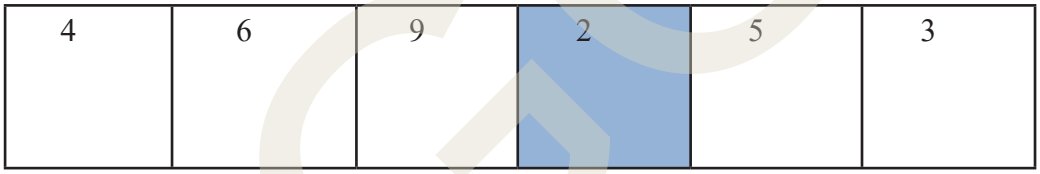
Step 2:



Key = 6; 9> 6, move 9 to the next position and insert key to the previous position

Result: [**4 6 9 2 5 3**]

Step 3:



Key = 2

9>2→[4 6 2 9 5 3]

6>2 →[4 2 6 9 5 3]

4>2 →[2 4 6 9 5 3]

Result : [**2 4 6 9 5 3**]

Step 4:

Key = 5

9 > 5 → [2 4 6 5 9 3]

6 > 5 → [2 4 5 6 9 3]

4 > 5 ≠ →Stop

Result: [**2 4 5 6 9 3**]

Step 5:

2	4	5	6	9	3

Key = 3

9>3 →[2 4 5 6 3 9]

6>3 →[2 4 5 3 6 9]

5>3 →[2 4 3 5 6 9]

4>3 →[2 3 4 5 6 9]

2>3 ≠→ Stop

Result: [**2 3 4 5 6 9**]

Now, let us consider our insertion sort example. Here, let us discuss the best case analysis first:

In the case of insertion sort, two operations are performed.

1. Scanning through the list, comparing each pair of elements
2. Swap elements if they are out of order.

As mentioned earlier, the best case refers to the minimum time the algorithm takes for 'n' input. The minimum time that the insertion sort can take is when the array is already in the sorted order. Therefore, in the best case, insertion sort runs in O(n) time.

Now let us consider **Worst and Average Case Analysis**:

The worst case for insertion sort will occur when the input list is in decreasing order.

In the case of an insertion sort with an input list in decreasing order, the following are the operations performed.

1. Scanning through the list, comparing each pair of elements → (1+2+---+n−2+n−1) scans
2. Swaps elements → (1+2+-----+n−2+n−1) swaps.

   i.e it takes 2*(1+2+-----+n−2+n−1) operations to perform insertion sort.

   →2*(n-1*(n-1+1))/2

   = n*n-1→ $O(n^2)$

Therefore, the worst-case complexity is $O(n^2)$.

When analysing algorithms, the average case often has the same complexity as the worst case. So, insertion sort, on average, takes $O(n^2)$. We will discuss in detail how to estimate the complexity in a short while.

4.1.5 Reasons to Analyse Algorithm

As with our pizza baking example, we can have multiple ways of preparing pizza. In the same manner, there may be multiple algorithms for a given problem. However, in order to determine which algorithm is more efficient than others, we have to analyse the algorithms. This analysis is done by comparing the time required for executing the algorithm (time complexity) or space required for executing the algorithm (space complexity). While considering time complexity, we consider the number of noticeable operations that are carried out by the algorithm.

The amount of time taken for completion of an algorithm is called the time complexity of the algorithm. It is a theoretical estimate that measures the growth rate of the algorithm T(n) for a large number of n, the input size. Usually, the time complexity is measured in terms of Asymptotic notations.

4.1.6 Asymptotic Notations

To communicate complex or large information efficiently and unambiguously, we use signs or symbols to represent ideas, concepts, or quantities in a concise and standardised way. This is called notation. To analyse algorithm performance, we use asymptotic notation.

Asymptotic Notations are languages that allow us to analyse an algorithm's running time by identifying its behaviour as the input size for the algorithm increases. Following are a few of the different types of Asymptotic Notations.

1. Big Oh - O-notation(Asymptotic Upper Bound)
2. Omega - Ω-notation(Asymptotic Lower Bound)
3. Big Theta - Θ-notation(Asymptotic Tight Bounds)

While analysing algorithms, we must consider what happens when the size of the input is large. We usually consider one algorithm to be more efficient if its worst-case running time has a lower order of growth. Algorithms are classified into three based on their order of growth. They are:

1. Algorithms that grow at least as fast as some function
2. Algorithms that do not grow fast
3. Algorithms that grow at the same rate

The above three categories are usually represented using the Asymptotic Notations Big Omega Ω (g(n)), Big Oh O(g(n)), and Big Theta Θ (g(n)) respectively. These notations will be discussed in detail in a short while.

4.1.6.1 Big Oh (O)notation(Asymptotic Upper Bound) - Worst-case

Big-O notation gives the worst-case complexity of an algorithm. It focuses on how the runtime of an algorithm scales with the size of the input.

If f(n) is the runtime of your algorithm and g(n) is a time complexity you are trying to relate to your algorithm, then f(n) is O(g(n)) if there exist real constants c (where c>0) and n0 such that: $f(n) \leq c \cdot g(n)$

For all input sizes n (where $n \geq n0$).

Consider the following functions

$f(n) = 3n + 2$

$g(n) = n$

We want to show that f(n)=O(g(n)).

$f(n) = 3n + 2$

Relate f(n) to g(n) , T(n) = 3*n + 2 <= 3*n + n <= 4 *n

Determine constants, $c = 4$ and $n_0 = 2$

Thus for $n \geq 2$, $3n+2 \leq 4n$

Worst-case Complexity: Big-O notation helps us understand the worst-case scenario for an algorithm's performance. In this example, as n grows large, the linear term 3n dominates the constant term 2. Therefore, the overall complexity can be simplified to O(n), indicating that the algorithm's runtime increases linearly with the input size.

Constants: The constants c and n0 are chosen to show that the inequality holds for sufficiently large input sizes. Here, c=4 and n0=2 work to demonstrate that 3n+2 is bounded above by 4n.

By using Big - Oh notation we can represent the time complexity as follows.

$3n + 2 = O(n)$

4.1.6.2 Big-Omega (Ω)-notation(Asymptotic Lower Bound) - Best case

Big-Ω notation provides the best case complexity of an algorithm. It focuses on how the runtime of an algorithm behaves in the best possible scenario as the input size grows.

If f(n) is the runtime of your algorithm and g(n) is a time complexity you are trying to relate to your algorithm, then f(n) is Ω(g(n)), if for some real constants c (where c > 0) and n0 (where n0 > 0), f(n) is >= c * g(n) for every input size n (where n > n0).

Consider the following functions

$f(n) = 3n + 2$

$g(n) = n$

If we want to represent f(n) as Ω(g(n)) then it must satisfy f(n) >= c*g(n) for all values of

c > 0 and n0 >= 1

f(n) >= c* g(n)

⇒ 3n + 2 >= c * n

3n+2 ≥ 3n Here, 3n is a lower bound for 3n+2 for n≥1.

Determine constants c and n0:

c = 3

n0 =1

Thus, for n≥1: 3n+2≥3n

Above condition is always TRUE for all values of c = 1 and n >= 1.

Best-case Complexity: Big-Omega notation helps us understand the best-case scenario for an algorithm's performance. In this example, as n grows large, the linear term 3n dominates the constant term 2. Therefore, the overall complexity can be simplified to Ω(n), indicating that the algorithm's runtime grows at least linearly with the input size.

Constants: The constants c and n0 are chosen to show that the inequality holds for sufficiently large input sizes. Here, c=3 and n0=1 work to demonstrate that 3n+2 is bounded below by 3n.

By using Big - Omega notation, we can represent the time complexity as follows

3n + 2 = Ω(n)

4.1.6.3 Big-theta (Θ)-notation(Asymptotic Tight Bounds) - Average-case

Θ-notation is used for analysing the average-case complexity of an algorithm. It gives a tight bound on the algorithm's runtime by defining both an upper and lower bound, indicating that the runtime grows asymptotically as a specific function of the input size in both the best and worst cases.

If f(n) is the runtime of your algorithm and g(n) is a time complexity you are trying to relate to your algorithm, then f(n) is Θ(g(n)) if there exist real constants c1, c2, and n0 (where c1>0, c2>0, n0>0) such that: c1*g(n) ≤ f(n) ≤c2.g(n) for all input sizes n (where n ≥ n0).

such that: c1.g(n)≤f(n)≤c2.g(n) for all input sizes n (where n≥n0).

f(n) is Θ(g(n)),

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

Relate f(n) to g(n):

To establish f(n) is Θ(g(n)), we need to find constants c1, c2, and n0

such that: c1.g(n)≤f(n)≤c2.g(n)

If we want to represent f(n) as Θ(g(n)) then it must satisfy C1 g(n) <= f(n) <= C2 g(n) for all values of C1 > 0, C2 > 0 and n0>= 1

C1 g(n) <= f(n) <= C2 g(n)

⇒ C1 n <= 3n + 2 <= C2 n

The above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 2.

if for some real constants c1, c2 and n0 (c1 > 0, c2 > 0, n0 > 0),

c1* g(n) is < f(n) is < c2 g(n) for every input size n (n > n0).

∴ f(n) is Θ(g(n)) implies f(n) is O(g(n)) as well as f(n) is Ω(g(n)).

Lower Bound: 3n+2≥3n, Here, c1=3.

Upper Bound: 3n+2≤3n+2n≤5n

Average-case Complexity: Big-Theta notation helps us understand the average-case scenario for an algorithm's performance. It tightly bounds the algorithm's runtime, indicating that it grows linearly with the input size, irrespective of minor variations.

Constants: The constants c1, c2, and n0 are chosen to show that the inequalities hold for sufficiently large input sizes. Here, c1=3, c2=5, and n0=1 work to demonstrate that 3n+2 is bounded both above and below by linear functions of n.

By using Big-Theta notation, we can represent the time complexity as follows.

3n + 2 = Θ(n)

The Fig.4.1.1 shows the graphical representation of Big-θ, Big-O and Big- Ω.

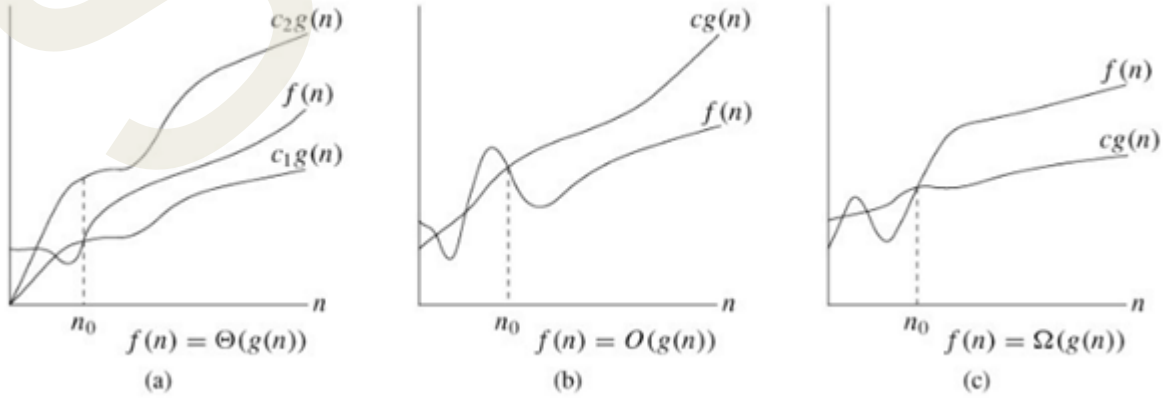


Fig.4.1.1 Graphical representation of asymptotic notations

Table 4.1.1 shows the main difference between the three notations.

Table 4.1.1 Difference between the three notations

Big oh (O)	**Big Omega (Ω)**	**Big Theta (Θ)**
Big oh (O) – Worst case	Big Omega (Ω) – Best case	Big Theta (Θ) – Average case
Big-O is a measure of the longest amount of time it could take for the algorithm to complete.	Big- Ω takes a small amount of time as compared to Big-O it could take for the algorithm to complete.	Big- Θ take a very short amount of time as compared to Big-O and Big-Ω it could take for the algorithm to complete.

4.1.7 Analyzing Algorithms

Basic algorithms are usually sequence, selection or iteration. In the case of a sequence, the order of those steps is crucial to ensuring the correctness of an algorithm. In the case of a selection, skipping of some sequences in an algorithm takes place based on the selection criteria. In the case of iteration, repetition takes place to execute steps a certain number of times or until a certain condition is met. This will become more clear in the following section.

Let us consider the following cases that a programmer uses to write an algorithm.

1. Simple statement
2. Sequence structure
3. The loop structure
4. If-then-else structure

4.1.7.1 Simple Statement

Let us assume that a statement takes a unit of time to execute, i.e., 1. Thus, T(n) = 1. The number of steps taken by the above algorithm for completion is one.

Thus **T(n) = 1** (1)

The above algorithm takes one step to complete.

Now, according to Big Oh notation (O-notation),

T(n) = O(g(n)) ,

T(n) <= c*g(n) where c > 0. (2)

To satisfy the relation 1, the relation 2 can be rewritten as:

T(n) <= 1*1

where c = 1 and n0 = 0. Comparing the above relation with relation 2, we get g(n) = 1.

Therefore, the above relation can be expressed as:

T(n) = O(1)

We can say that the time complexity of the above algorithm is O(1), i.e., of the order 1

expressed as:

T(n) = O(1)

Let us take another example where time complexity is more than one.

Example 1: compute the time complexity of the following relation

T(n)= 254

Solution

From relation 2,

T(n) <= c*g(n) where c > 0

From example 1,

- T(n)= 254
- T(n)<=254*1
- c=254, g(n)= 1
- T(n)=O(1)

The above example is the case with sequence structure, the details of which are given below.

4.1.7.2 Sequence Structure

Here, the execution time of the sequence structure is equal to the sum of the execution time of individual statements present within the sequence structure. Consider the following algorithm. It consists of a sequence of statements 1 to 4:

Let us consider the following algorithm to calculate the area of a rectangle.

Algorithm Area Rectangle

{

Step

1. Read length, breadth;

2. Area = length * breadth;

3. Print Area;

4. Stop

}

The number of steps taken by the above algorithm for completion is four.

Thus **T(n) = 4**

The above algorithm takes four steps to complete.

Now, according to Big Oh notation (O-notation)

T(n) = O(g(n)) ,

T(n) <= c*g(n) where c > 0. (2)

To satisfy the relation 2, the relation can be rewritten as:

T(n) <= 4*1 where c = 4 and n0 = 0

Comparing the above relation with relation 2, we get g(n) = 1. Therefore, the above relation can be expressed as:

T(n) = O(1)

We can say that the time complexity of the above algorithm is O(1), i.e., of the order 1

expressed as:

T(n) = O(1)

4.1.7.3 The loop structure

Let us consider the following algorithm to find the number of ones.

```
Algorithm Count_ones()
{
    Step
    1. Num_of_ones = 0;
    2. For (I = 1 to N)
    {
        2.1 Read Num;
        2.2 If (Num == 1)
            Num_of_ones = Num_of_ones +1;
    }
    3. Prompt "The number of ones elements =";
    4. Print Num_of_ones;
    5. Stop
}
```

This algorithm takes the following steps for its completion:

No. of simple steps = 4 (Steps 1, 3, 4, 5)

No. of Loops of steps 1 to N = 1

No. of statements within the loop = 3 (1 comparison, 1 addition, 1 assignment)

Thus, T(n) = 3*N + 4

Now, for N >= 4, we can say that

3*N + 4 <= 3*N + N <= 4N

Therefore, we can say that

T(n) <= 4N where c = 4, n0 >= 4

T(n) = O(N)

Hence, the given algorithm has a time complexity of the order of N, i.e., O(N). This indicates that the term 4 has negligible contribution in the expression: 3*N + 4.

Note: In the case of nested loops, the time complexity depends upon the count of both outer and inner loops. Consider the nested loops given below. This program segment contains 'S', a sequence of statements within nested loops I and J.

```
For (I = 1; I <= N; I++)
{
        For (J = 1; J <= N; J++)
        {
                S;
        }
}
```

It may be noted that for each iteration of the I loop, the J loop executes N times. Therefore, for N iterations of the I loop, the J loop would execute N*N = N^2 times. Accordingly, the statement S would also execute N^2 times. Thus,

$T(n) = N^2$

To satisfy the relation 2, the above relation can be rewritten as shown below:

$T(n) <= 1 * N^2$

where c = 1 and n0 > 0. Comparing the above relation with relation 2, we get $g(n) = N^2$. Therefore, the above relation can be expressed as:

$T(n) = O(N^2)$

Hence, the given nested loop has a time complexity of the order of N^2, i.e., $O(N^2)$.

Now, let us calculate the time complexity of the following questions.

Compute the time complexity of the following relations:

(1) T(n) = 2834

(2) T(n) = 9*n + 18

(3) T(n) = 18*(N^2) + 7

(4) T(n) = 8*(N^3) +3 N^2 + 6*N

Solution:

(1) T(n) = 2834 <= 2834*1, where c = 2834 and n_0 = 0.

= O(1)

Ans. The time complexity = O(1)

(2) T(n) = 9*n + 18 <= 9*n + n <= 10 *n for c = 10 and n_0 = 18

= O(N)

Ans. The time complexity = O(N)

(3) T(n) = 18*N^2 + 7 <= 18*N^2 + N for N = 7

Now, for N <= N^2, we can rewrite the above relation as given below:

18*N^2 + N <= 18*N^2 + N^2 <= 19 N^2 for c = 19 and n0 = 7

Thus, T(n) = O(N^2)

Ans. The time complexity = O(N^2)

(4) T(n) = 8*(N^3) + 3N^2 + 6*N

For N^2 >= 6*N, we can rewrite the above relation as given below:

8*N^3 + 3N^2 + 6*N <= 8*N^3 + 3N^2 + N^2 <= 8*N^3 + 4*N^2

For N^3 >= 4*N^2, we can rewrite the above relation as given below:

8*N^3 + 4*N^2 <= 8*N^3 + N^3 <= 9*N^3 for c = 9 and n0 = 6

Thus, T(n) = O(N^3)

Ans. The time complexity = O(N^3)

4.1.7.4 If-then-else structure

In the if-then-else structure, the then and else parts are considered separately. Consider the if-then-else structure given in Figure 4.1.1. Let us assume that the 'statement 1' of the part has the time complexity T1 and the 'statement 2' of the else part has the time complexity T2. The time complexity of the if-then-else structure is taken to be the maximum of the two, i.e., max(T1, T2).

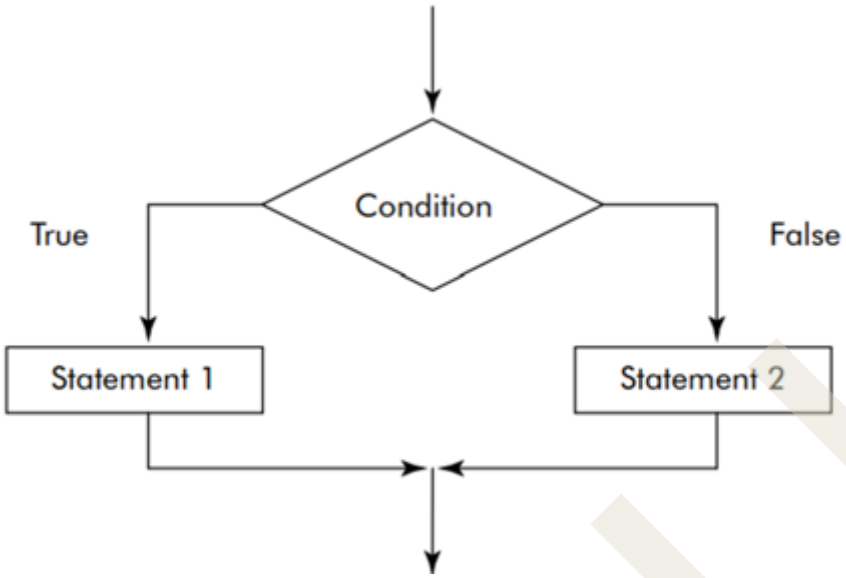


Fig 4.1.1 the if-then-else structure

Example 10: Consider the following algorithm. Compute its time complexity.

```
If (a > b)
 {
 a = a - 1;
 }
else
 {
        for (i=1; i <= N, i++)
        {
         a = a+i;
        }
 }
```

Solution: It may be noted that in the above algorithm, the time complexity of the 'then' and 'else' parts are O(1) and O(N), respectively. The maximum of the two is O(N). Therefore, the time complexity of the above algorithm is O(N).

Recap

- Essential Properties of algorithms
- Cases to consider during analysis
  - Best case
  - Worst case
  - Average case
- Complexity of algorithms
  - Space complexity
  - Time complexity
- Asymptotic notations
  - Big - O
  - Big - Ω
  - Big - Θ
- Differences between the three asymptotic notations

Basic Algorithms:

- Sequence: The order of steps is crucial.
- Selection: Skipping steps based on criteria.
- Iteration: Repetition until a condition is met.

Objective Type Questions

1. In the case of an algorithm, the _____ refers to zero or more quantities that are externally supplied.
2. In the case of an algorithm, _____is at least one quantity produced.
3. _____is the property that each instruction of an algorithm is clear and unambiguous.
4. The property that the algorithm terminates after a finite number of steps is known as _____.

5. _____ of an algorithm means that each instruction should do some task.
6. Big O is a measure of _____ time complexity.
7. Big Ω is a measure of _____time complexity.
8. Big Θ is a measure of _____ time complexity
9. Define asymptotic notation.
10. What does Big O notation represent?
11. How is the execution time of a sequence structure determined?
12. Explain the significance of ignoring constant coefficients in time complexity analysis.
13. What is the time complexity of a nested loop where both loops run from 1 to N?

Answers to Objective Type Questions

1. Input
2. Output
3. Definiteness
4. Finiteness.
5. Effectiveness
6. Worst case
7. Best case
8. Average case
9. Mathematical tools to describe algorithm efficiency and scalability.
10. The worst-case running time of an algorithm.
11. Sum of the execution times of individual statements.
12. To focus on the most significant part of an algorithm's running time.
13. $O(N^2)$

Assignments

1. Compute the time complexity of the following relations:

   i. $T(n) = 410$

   ii. $T(n) = 4*n - 12$

   iii. $T(n) = 13*N^2 - 7$

   iv. $T(n) = 3*N^3 + N^2 + 4*N$

2. What is the time complexity of the following loop?

```
for (int i = 0; i < n; i++)
  {
    statement;
  }
```

3. What is the Big-Oh complexity of the following nested loop?

   i.
```
for (int i = 0; i < n; i++)
    {
    for (int j = 0; j < n; j++)
        {
        statement;
        }
    }
```

   ii.
```
for (int i = 0; i < n; i++)
    {
    for (int j = i + 1; j < n; j++)
     {
        statement;
     }
    }
```

iii.
```
for (int i = 0; i < n; i++)
    {
     for (int j = n; j > i; j--)
        {
          statement;
        }
    }
```

iv.
```
for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < m; j++)
       {
          statement1;
       }
       statement2;
   }
```

4. Consider the following algorithm and compute its time complexity.

```
if (a==b)
        statement1;
 else if (a > b)
       statement2;
 else
       statement3;
```

5. Explain the different complexities of algorithms.

6. Explain Asymptotic notations.

Suggested Reading

1. Data Structures and Algorithms, A. Aho, J. Hopcroft, and J. Ullman, Computer Science and Information Processing Addison-Wesley, Reading, Massachusetts, 1st edition, (Jan 11, 1983)

2. “Data structures Using C” by A K Sharma

3. “Data Structures and Program Design in C” by Kruse Robert L

4. “Data Structures and Algorithm Analysis in C” by Mark Allen Weiss

5. Tenenbaum, Aaron M. Data structures using C. Pearson Education India, 1990.

6. Horowitz, Ellis, Sartaj Sahni, and Susan Anderson-Freed. Fundamentals of data structures. Vol. 1982. Potomac, MD: Computer Science Press, 1976.

UNIT 2

Searching and Sorting

Learning Outcomes

By the completion of this unit, the learner will be able to:

- introduce the concept of searching and sorting in data structures
- familiarise various sorting and searching algorithms
- make them understand the concept of time complexity analysis for various algorithms
- describe the estimation of space complexity for various sorting algorithms

Prerequisites

There are many operations we can perform on a collection of things. In our daily lives, we also handle similar arrangements, such as stacks of plates, closets of clothes, and books on shelves. We regularly need to process these types of arrangements, particularly when searching for specific items or sorting them in a certain order. For example, we might search for our clothes or organise books every day. Here, we will discuss these methods and explain how we can process them easily and efficiently.

Array

Array is a collection of similar elements having the same datatype, accessed using a common name. Array elements occupy contiguous memory locations. Every component of an array is assumed to have some indices starting at 0 and ending at n-1, where n is the size of the array. An example of arrays is shown in Fig 4.2.1(a).

15	20	32	45	23	50	40
0	1	2	3	4	5	6

Fig 4.2.1(a)

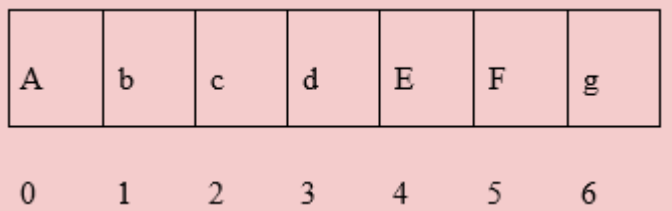


Fig 4.2.1(b)

Figure 4.2.1(a) is a visualisation of an integer array, and Figure 4.2.1(b) is a character array.

Declaring a one-dimensional Array using C

Syntax:

element type ***array_name*** [size_array]

Where

- Element type is any type that already exists
- Array name is the name of the array
- Size_array is the number of elements in the array

This declares a named array whose elements will be of type element type and that has size_array many elements.

Examples

```
char fname[24]; // an array named fname with 24 characters

int grade[35]; // an array named grade with 35 integers
```

Complexity

In the context of computers, complexity is a term that is used to specify computational complexity. Computational complexity is a characterisation of time or space requirements for solving a problem by a particular algorithm. There are two types of computational complexity: time complexity and space complexity. The time complexity of a program is the amount of computer time that it needs to run to completion. The space complexity of a program is the amount of memory that it needs to complete.

Key Concepts

Linear Searching, Binary search, Sorting, Selection Sort and Insertion sort

Discussion

4.2.1 Searching

Searching involves finding a specific item within a set of given elements. In our daily lives, we search for various things in different ways to make access easier. For example, I might look for a book on a shelf, a word in a dictionary, or clothes in a closet. Each of these searches uses different techniques based on the context. Ultimately, the best methods are those that save time and are most efficient for the task at hand. The search is successful if the required element is found. Otherwise, the search is unsuccessful.

The most commonly used searching techniques are

- Linear search
- Binary search

4.2.2 Linear search

Suppose we need to find a specific book in an unsorted stack of books. You begin at the top of the stack and examine each book sequentially until you find the one you're looking for. This is an example of linear search, where each element is checked one by one.

In computer science, a linear search or sequential search is a method for finding an element within a list, as visualised in Figure 4.2.2. It sequentially checks each element of the list until a match is found or the whole list has been searched. The algorithm used for linear searching is given below.

Fig 4.2.2 Linear search

4.2.2.1 Algorithm

```
Linear_Search (a, n, item, loc)
        # 'a' is the given array
        # 'n' is the number of elements in the  array a
```

```
        # 'item' is the item to be searched
        # 'loc refers to the location or index of a searched element
Begin
   for i = 0 to (n - 1) by 1 do
      if (a[i] = item) then
         set loc = i
         Exit
      End if
   End for
   set loc = -1
End
```

Consider an array ARR={15,20,32,45,23,50,40}

In the array, there are 7 elements. Suppose we want to check whether element 23 is present in the array or not. So, we need to search for element 23 in the array. During searching, either we will find the element at some index position i<7, or we will reach the end of the array without finding the element 23. These are the two possibilities of how the searching process gets terminated.

4.2.2.2 Time complexity of linear search

a. Best case

Suppose we want to search for the first element. In that case, we need to perform only one comparison. So, the time complexity is O(1).

b. Worst case

Suppose we want to search the last element in the list. In that case, we need 'n' number of comparisons where 'n' is the size of the array. Therefore, the time complexity is O(n).

c. Average case

Usually, the average case is calculated as the ratio between the sum of the total number

of different cases and the number of elements. Here, for the searching of the first element, we need 1 comparison; for the second element, we need 2 comparisons, and so on. Thus, for searching for the nth element, we need 'n' comparisons. So, the time complexity is

$$(1+2+3+\cdots+n)/n = (n\,((n+1))\,/\,2)/n = (n+1)/2$$

$$\mathbf{O((n+1)/2) \approx O(n)}$$

4.2.2.3 Example for Linear Search

Consider the following array shown in Fig 4.2.3. It is an integer array of 7 elements. Suppose we want to search for element 25.

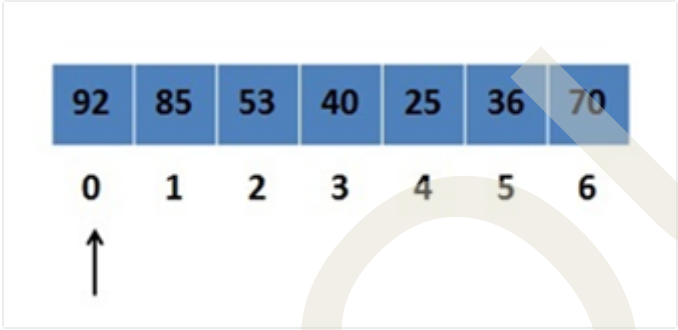


Fig 4.2.3 Array of integers

Now, the linear search algorithm compares element 25 with all elements of the array one by one. It continues searching until either element 25 is found or reaches the end of the array. Let us analyse the working of the search algorithm step by step. Assume there exists an array pointer. Initially, it points at index 0.

Step 1:

It compares element 25 with the 1st element 92. Since 25 ≠ 92, the required element has not been found. So, it moves to the next element, as shown in the figure below 4.2.4.

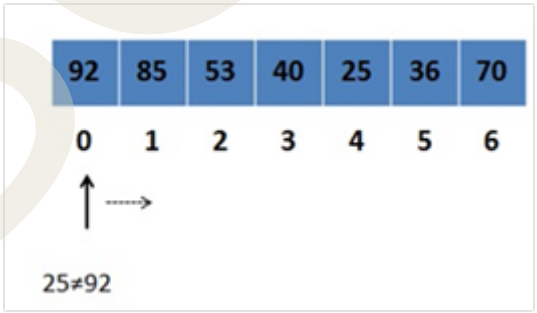


Fig 4.2.4 Array of integers

Step 2:

It compares element 25 with the 2nd element 85. Since 25 ≠ 85, the required element has not been found. So, it moves to the next element as shown in the figure below 4.2.5.

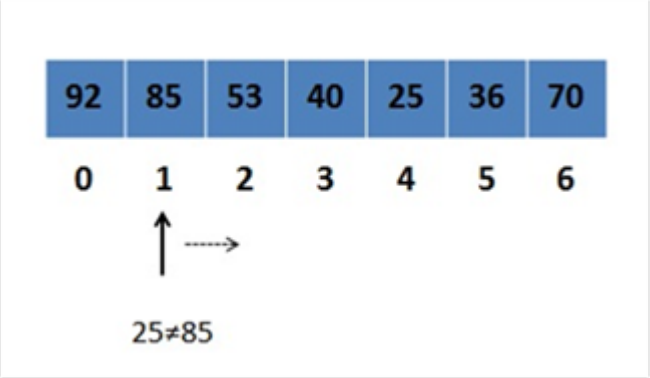


Fig 4.2.5 Array of integers

Step 3:

It compares element 25 with the 3rd element 53. Since 25 ≠ 53, the required element is not found. So, it moves to the next element.

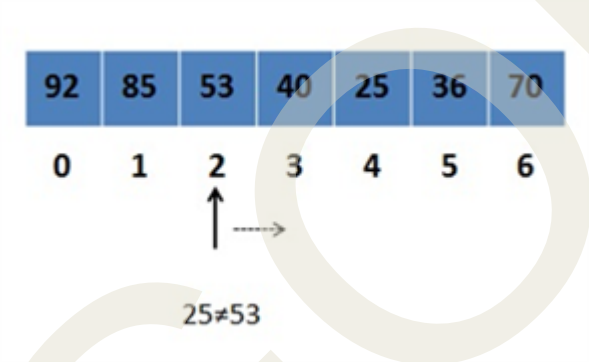


Fig 4.2.6 Array of integers

Step 4:

It compares element 25 with the 4th element 40. Since 25 ≠ 40, the required element is not found. So, it moves to the next element.

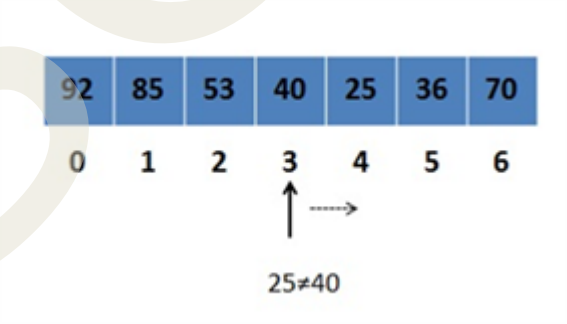


Fig 4.2.7 Array of integers

Step 5:

It compares element 25 with the 5th element 25. Since 25 = 25, the required element is found. Now, it stops the comparison and returns index 4, at which element 25 is present.

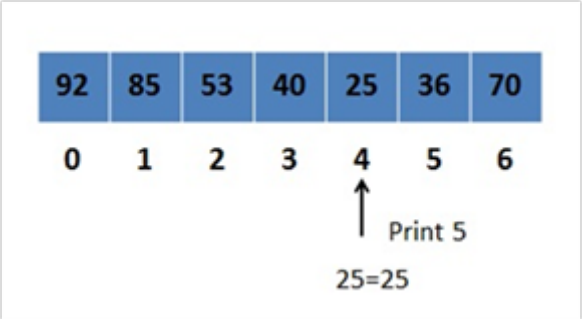


Fig 4.2.8 Array of integers

4.2.3 Binary Search

Have you ever used a dictionary? How do you search for a word in it? You open the dictionary to the middle page and compare your target word with the words on that page. If your target word comes before the middle word, you search the first half; if it comes after, you search the second half. You continue this process until you find the word.

Binary search is an efficient algorithm for finding an item from a sorted list of elements. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

4.2.3.1 Algorithm

```
function binarySearch(array, target):
    low = 0
    high = length(array) - 1
    while low <= high:
        mid = low + (high - low) / 2
        If array[mid] == target:
            return mid  // Target found, return its index
        else if array[mid] < target:
            low = mid + 1  // Target is in the upper half
        else:
            high = mid - 1  // Target is in the lower half
    return -1  // Target not found
```

4.2.3.2 Example for Binary Search Algorithm

The binary search algorithm efficiently finds the target value in a sorted array by repeatedly dividing the search interval in half.

Consider an array containing 7 elements, as shown in Fig 4.2.9

Fig 4.2.9 Array of integers

Let x = 4 be the element to be searched.

Set two pointers low and high at the lowest and the highest positions, respectively, as shown in Fig 4.2.10

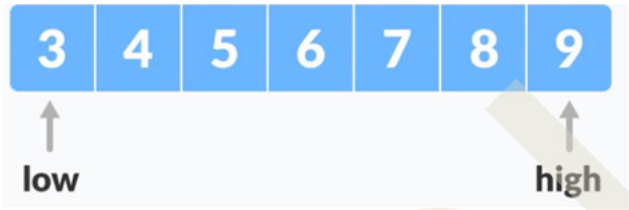


Fig 4.2.10 Array of integers

Find the middle element mid of the array ie. arr[(low + high)/2] = 6

Fig 4.2.11 Array of integers

If x == mid, then return mid.

else, compare the element to be searched with mid.

if x > mid, compare x with the middle element of the elements on the right side of mid.

This is done by setting low to low = mid + 1.

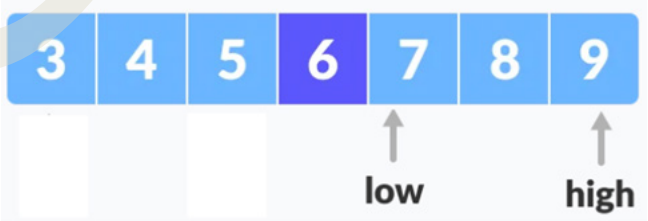


Fig 4.2.12 Array of integers

Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to high = mid - 1.

Fig 4.2.13 Array of integers

Repeat steps 3 to 6 until low meets high.

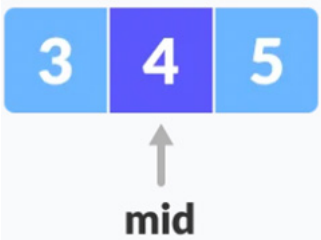


Fig 4.2.14 Array of integers

x=4 is found

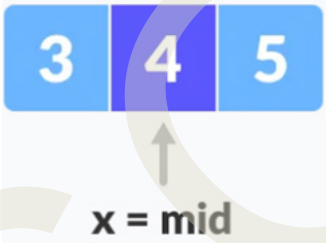


Fig 4.2.15 Array of integers

4.2.3.3 Time Complexity of Binary Search

The time complexity of the binary search algorithm can be analysed in terms of best case, worst case, and average case scenarios.

a. Best Case Time Complexity

The target element is found in the middle of the array on the first comparison. In the best case, binary search finds the target element in the first comparison itself, requiring only one operation. So, the time complexity is O(1).

b. Worst Case Time Complexity

The target element is not in the array or is located at the very end of the search process. In the worst case, the search space is halved each time, leading to the maximum number of comparisons being equal to the logarithm (base 2) of the number of elements in the array. Thus, the time complexity is $O(\log_2 n)$.

c. Average Case Time Complexity

The target element is found at some point during the search but not necessarily at the first comparison. On average, the binary search will need to reduce the search space

logarithmically. Hence, the average time complexity is also $O(\log_2 n)$.

4.2.4 Sorting

Observe the picture below. The boys are systematically arranging the books on a shelf, or we can say they are sorting the books. The criteria used for sorting may be different. It may be according to the author of the book, according to the subject like science, arts, etc. Sorting is nothing but arranging a group of things systematically according to certain criteria.

Fig 4.2.16 Sorting books

In this digital era, there may be situations where digital data needs to be sorted. For example, we have a list of student data, and we want to sort them according to their age. There, we apply different algorithms to arrange the data systematically. The algorithms are called sorting algorithms. Here, we discuss two sorting algorithms: selection sort and insertion sort.

4.2.5 Selection Sort

A teacher organises children for seating by selecting the smallest child from a group and seating them in the first row. Then, the teacher picks the second smallest child, seats them next to the first one, and continues this process until all children are seated.

Selection sort is one of the easiest approaches to sorting. It is inspired by the way in which we sort things out in day-to-day life. It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting. Suppose we want to sort a list of integers.

The logic of Selection sort for sorting a list of integers is given below.

Step1. First, find the smallest element on the given list.

Step2. Swaps it with the first element of the unordered list.

Step3. Find the second smallest element.

Step4. Swaps it with the second element of the unordered list.

Similarly, it continues to sort the given elements.

4.2.5.1 Algorithm

```
# index is a variable to store the index of the minimum element
# j is a variable to traverse the unsorted sub-array
# temp is a temporary variable used for swapping

for (i = 0 ; i < n-1 ; i++)
{
    index = i;
    for(j = i+1 ; j < n ; j++)
    {
        if (A[j] < A[index])
        index = j;
     }
     temp = A[i];
     A[i] = A[index];
     A[index] = temp;
}
```

4.2.5.2 Selection Sort Example

Consider the integer array given below in Fig 4.2.17. Suppose we want to sort the array elements in ascending order.

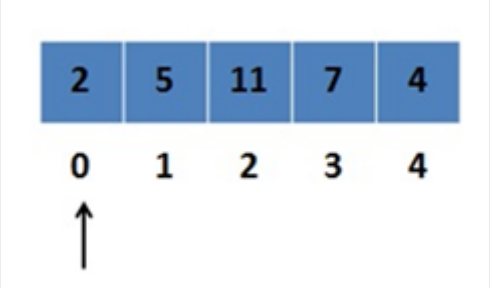


Fig 4.2.17 Array of integers

Step1: for i = 0

Compare the first element with all other elements in the array to find out the smallest element and swap it with the first element in the array.

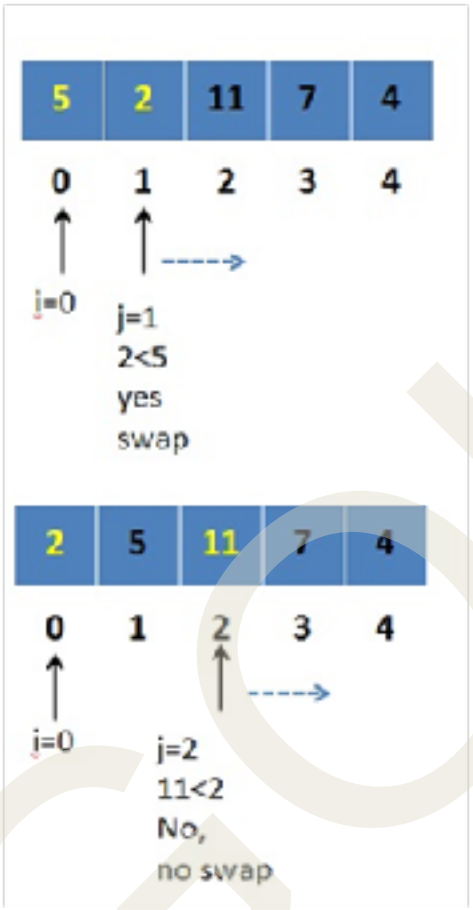


Fig 4.2.18 Array of integers

Finally, after comparing the last element in the array, we will get the element at the index 1. that is, 2 is the smallest element. So, swap 2 and element in the first position, i.e., swap 2 and 5. Now, the array will be partially unsorted as follows.

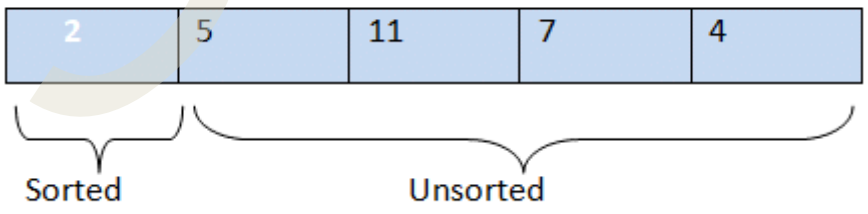


Fig.4.2.19 Array of integers

Step 2: for i=1

Find the minimum element in the unsorted array, comparing the second element with

all other elements in the unsorted array and swap it with the second element of the array. Observe the pictorial representation to understand the logic.

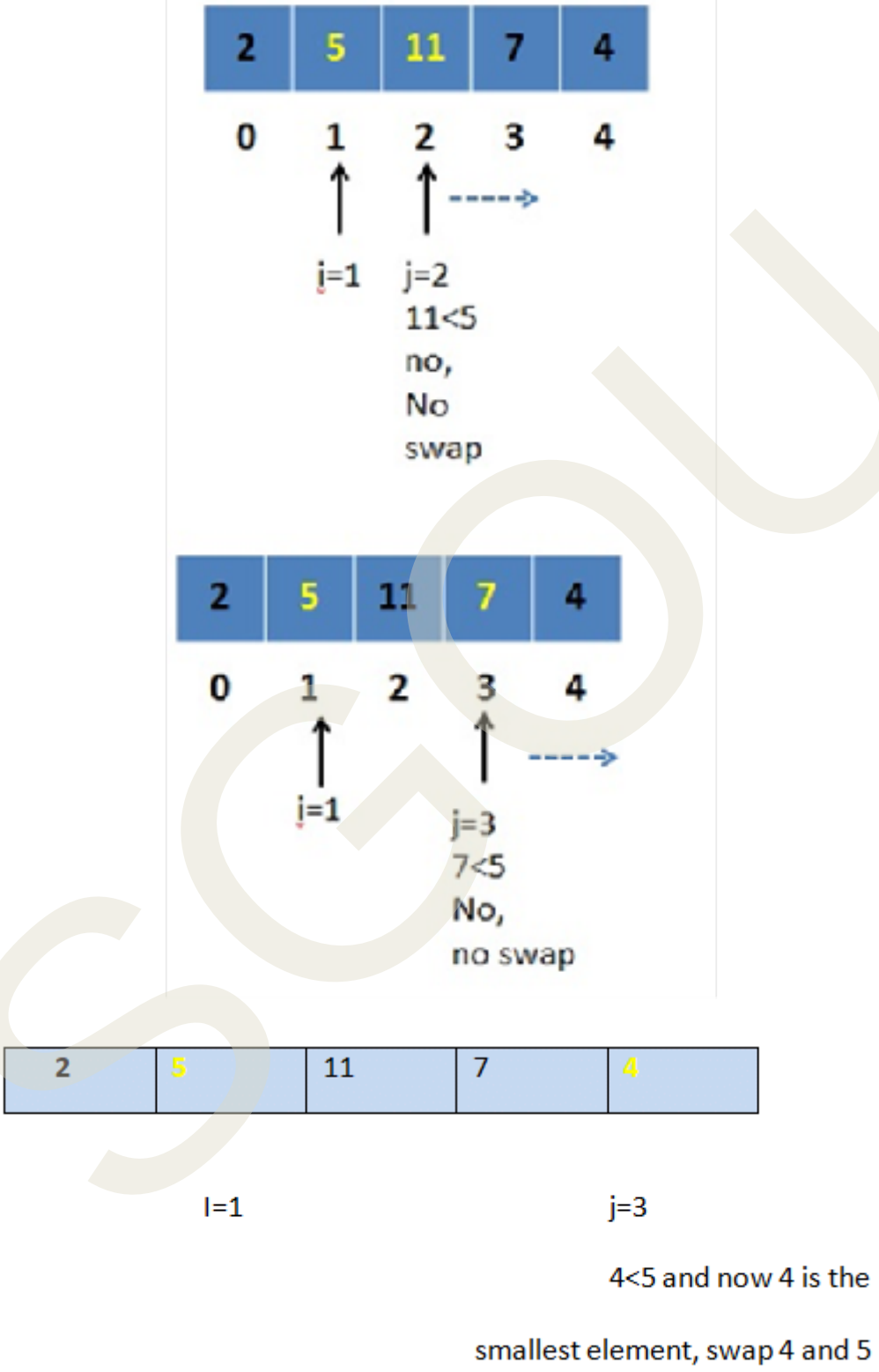


Fig 4.2.20 Array of integers

After comparing the second element of the array with all other elements, the array will be partially unsorted as follows.

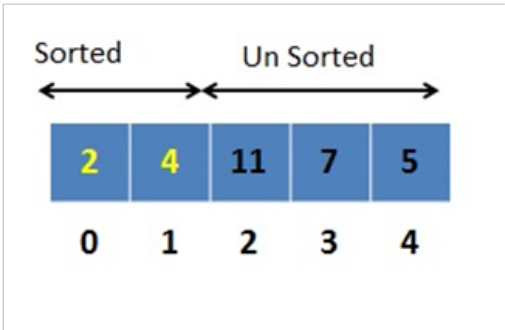


Fig 4.2.21 Array of integers

Step 3: for i=2

Find the minimum element in the unsorted array, compare the third element with all other elements in the unsorted array, and swap it with the third element of the array. Observe the pictorial representation to understand the logic.

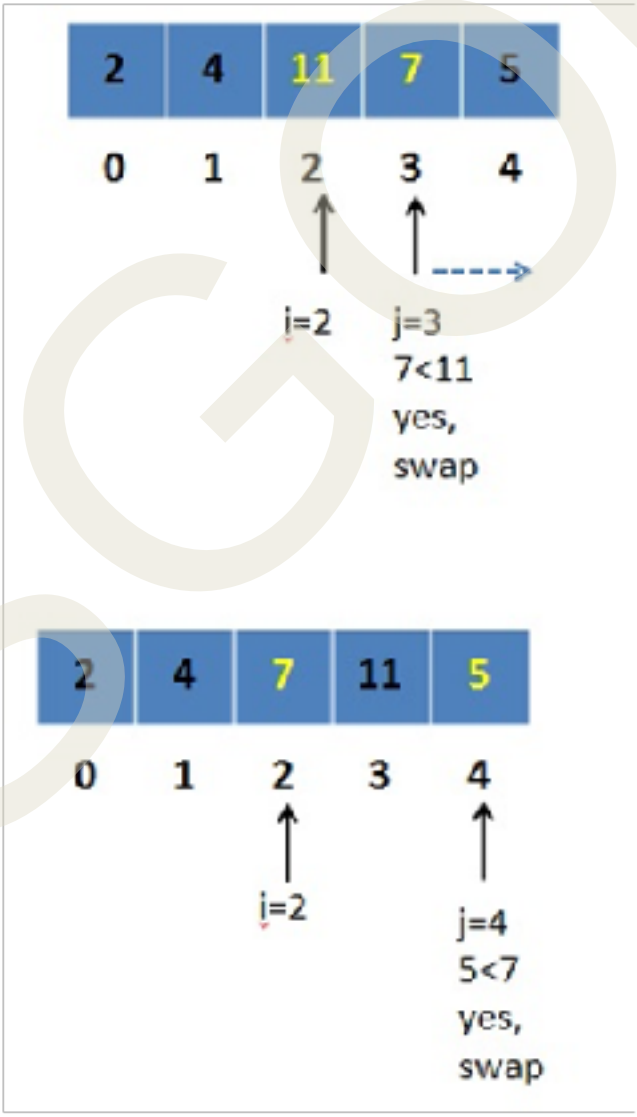


Fig 4.2.22 Array of integers

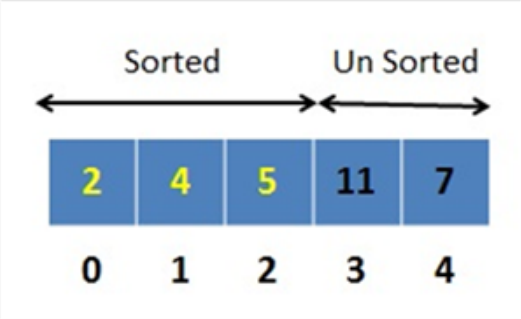


Fig 4.2.23 Array of integers

Step 4: For i=3

Find the minimum element in the unsorted array, compare the fourth element with all other elements in the unsorted array, and swap it with the fourth element of the array. Observe the pictorial representation to understand the logic.

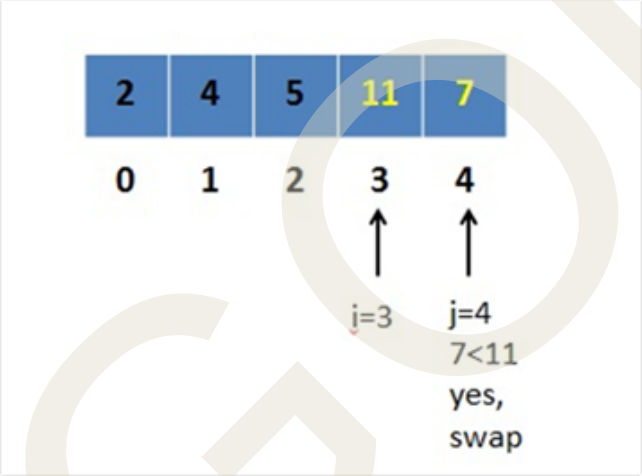


Fig 4.2.24 Array of integers

After comparing the second element of the array with all other elements, the array will be sorted as follows.

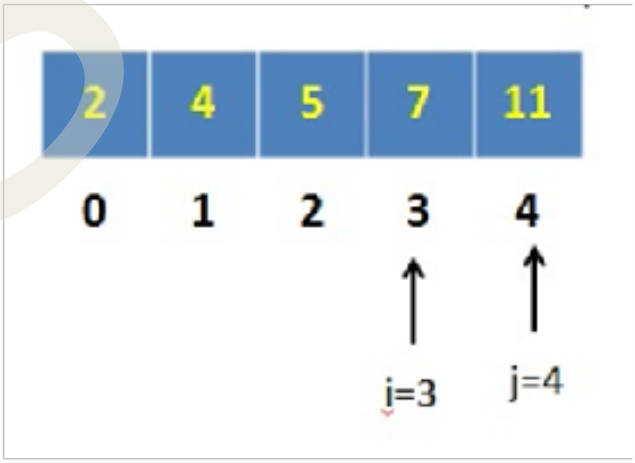


Fig 4.2.25 Array of integers

Step 5:for i=4

Loop gets terminated as 'i' becomes 4. The state of the array after the loops are finished is shown in the figure below.

Fig 4.2.26 Array of integers

So, we can conclude that in selection sort, with each loop cycle, the minimum element in an unsorted subarray is selected. It is then placed at the correct location in the sorted subarray until the array is completely sorted.

4.2.5.3 Time Complexity of Selection Sort

Selection sort algorithm consists of two nested loops. Owing to the two nested loops, it has $O(n^2)$ time complexity for the best case, average case and worst case.

> Selection sort is not a very efficient algorithm when data sets are large. The average and worst case complexities indicate this. Selection sort uses a minimum number of swap operations O(n) among all the sorting algorithms.

4.2.6 Insertion sort

Insertion sort is a simple sorting algorithm. It works similar to the way we sort playing cards in our hands. In insertion sort, the array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part. For a given array **'A'** of size **'n'**, to sort the array in ascending order, the insertion sort works as follows:

Step 1: Iterate from **A[0]** to **A[n-1]** over the array.

Step 2: Compare the current element (key) to its predecessor.

Step 3: If the key element is smaller than its predecessor, compare it to the elements before.

Move the greater elements one position up to make space for the swapped element.

4.2.6.1 Algorithm

```
#A is the given array
# variable to traverse the array A
#key = variable to store the new number to be inserted  into the sorted sub-array
# j = variable to traverse the sorted sub-array
for (i = 1 ; i < n ; i++)
{
        key = A [ i ];
        j = i - 1;
        while(j > 0 && A [ j ] > key)
        {
            A [ j+1 ] = A [ j ];
            j--;
        }
       A [ j+1 ] = key;
}
```

4.2.6.2 Insertion sort example

Consider the array elements 7, 3, 11, 8 and 5. The steps needed to sort the elements in ascending order are given below.

Step 1: for i=1

Fix the 2nd element as the key element. Compare it with all elements of a sorted array. If it is smaller, then place the element in the correct position.

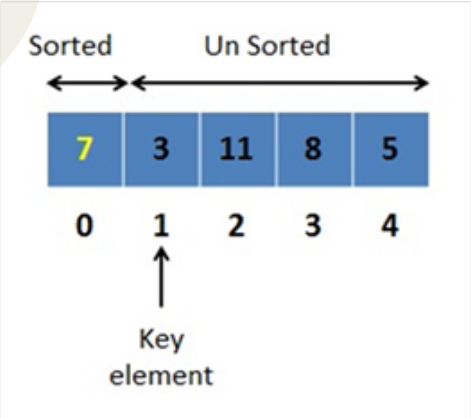


Fig 4.2.27 Array of integers

In the given array, since 3<7, it shifts 7 towards the right and places 3 before it. So the resulting list becomes as follows.

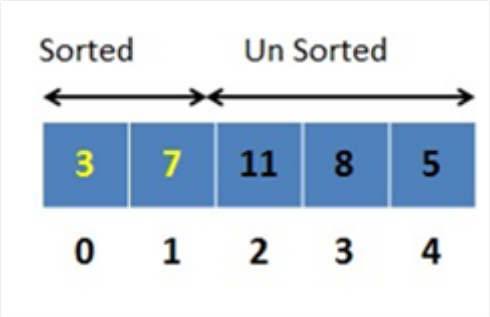


Fig 4.2.28 Array of integers

Step2 : for i=2

Fix the 3rd element as the key element. Compare it with all elements of a sorted array. If it is smaller, then place the element in the correct position.

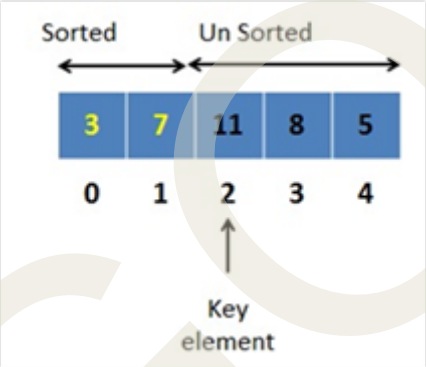


Fig 4.2.29 Array of integers

Here, for i=2, the key element, 11> (3,7). So, no shifting takes place. The resulting list remains the same, and it is shown below in Fig 4.2.28.

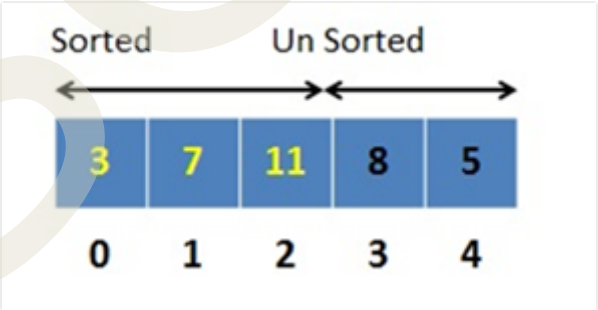


Fig 4.2.30 Array of integers

Step 3: for i=3

Fix the 4th element as the key element. Compare it with all elements of a sorted array. If it is smaller, then place the element in the correct position.

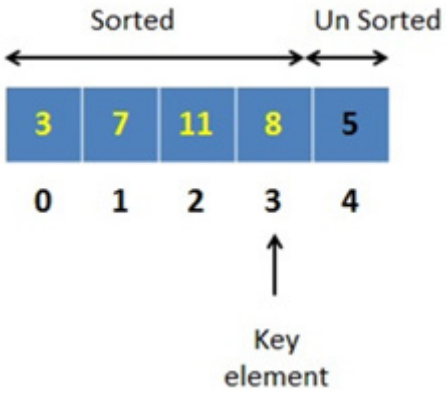


Fig 4.2.31 Array of integers

Here, for i=3, the key element, 8< 11. So it shifts 11 towards the right and places 8 before it. The resulting list is given below 4.2.32.

Step 5:for i=4

Fix the 5th element as the key element. Compare it with all elements of a sorted array. If it is smaller, then place the element in the correct position.

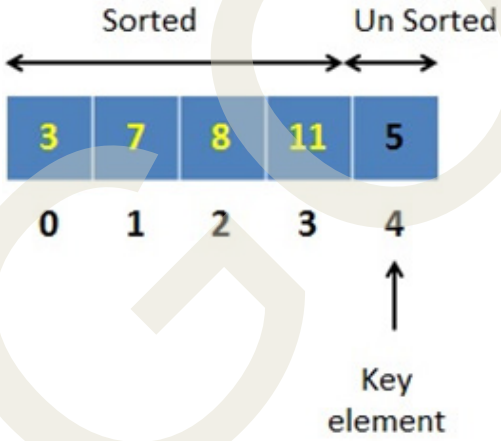


Fig 4.2.32 Array of integers

Here, for i=4, the key element, 5< (7,8,11). So it shifts (7,8,11) towards the right and places before it. The resulting list is given below.

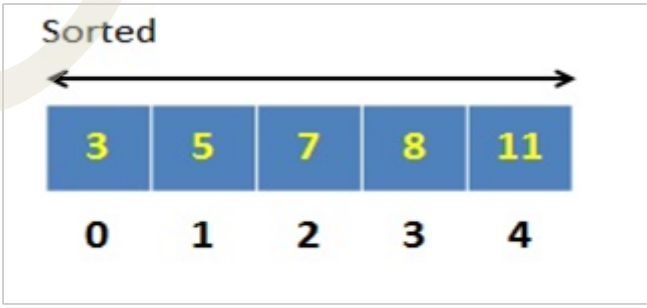


Fig 4.2.33 Array of integers

Loop gets terminated as 'i' becomes 5. The state of the array after the loops are finished is shown in the figure above.

4.2.5.2 Time and space complexity

The best case input is an array that is already sorted. In this case, in each iteration of insertion sort, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. The worst case time complexity of insertion sort is of $O(n^2)$. The space complexity of insertion sort is O(1).

Recap

- Linear Search Algorithm is the simplest searching algorithm.
- Linear Search Algorithm searches for an element by comparing it with each element of the array.
- Time complexity of linear search
  - Best case=O(1)
  - Average Case=O((n+1)/2)
  - Worst case=O(n)
- Selection sort is an in place sorting algorithm.
- Selection Sort is the easiest approach to sorting.
- Selection Sort Algorithm Worst case Time Complexity is $O(n^2)$.
- Selection Sort Algorithm Space Complexity is O(1).
- In insertion sort, the array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed in the correct position in the sorted part.
- The best case time complexity of insertion sort is O(n).
- The worst case time complexity of insertion sort is $O(n^2)$.

Objective Type Questions

1. Where is linear searching used?
2. What is the average number of key comparisons done in a successful sequential search in a list of length n?
3. What is the best case time complexity of the linear search algorithm?
4. What is the worst case time complexity of the Linear Search algorithm?

5. What is the disadvantage of linear searching?
6. What is the space complexity of selection sort?
7. What is the worst case time complexity of the selection sort algorithm?
8. What is the best case time complexity of insertion sort?
9. What is the worst case time complexity of insertion sort?

Answers to Objective Type Questions

1. When the list has only a few elements
2. O((n+1)/2)
3. O(1)
4. O(n)
5. Greater complexity (in worst case)
6. O(1)
7. $O(n^2)$
8. O(n)
9. $O(n^2)$

Assignments

1. Explain different searching techniques commonly used in computers.
2. Write an algorithm for linear search and binary search.
3. What is sorting? What are the techniques used for sorting?
4. Explain the selection sort using an algorithm and an example.
5. What are the disadvantages of the selection sort algorithm?
6. Explain the insertion sort algorithm with examples.
7. What is the difference between searching and sorting an algorithm?

Suggested Reading

1. Data Structures and Algorithms, A. Aho, J. Hopcroft, and J. Ullman, Computer Science and Information Processing Addison-Wesley, Reading, Massachusetts, 1st edition, (Jan 11, 1983)

2. Kruse, Robert, and C. L. Tondo. Data structures and program design in C. Pearson Education India, 2007.

3. Tenenbaum, Aaron M. Data structures using C. Pearson Education India, 1990.

4. Horowitz, Ellis, Sartaj Sahni, and Susan Anderson-Freed. Fundamentals of data structures. Vol. 1982. Potomac, MD: Computer Science Press, 1976.

Divide and Conquer Algorithms & Backtracking Algorithms

Learning Outcomes

By the completion of this unit, the learner will be able to:

- describe how to design a divide and conquer algorithms
- introduce quicksort
- familiarise binary search
- make aware of the minimax algorithm
- familiarise Prim's algorithm and Kruskal's algorithm

Prerequisites

When solving problems, you may have encountered situations where breaking down a large task into smaller, more manageable tasks makes it easier to handle. This is similar to organising your study sessions by splitting your subjects into smaller topics and tackling them one at a time. This approach helps you focus better and understand each part more thoroughly.

In computer science, a similar strategy is used, called the Divide and Conquer algorithm. This method involves breaking down a complex problem into smaller, simpler sub-problems, solving each of these sub-problems, and then combining their solutions to solve the original problem. This technique is very effective in solving a variety of computational problems and is used in many well-known algorithms such as Merge Sort, Quick Sort, and Binary Search.

Understanding the Divide and Conquer approach will help you appreciate how powerful and efficient it can be in solving problems that might seem too complicated at first glance. By learning this technique, you will gain insight into how large problems can be simplified and solved step-by-step, which is a valuable skill not just in computer science but in many areas of life.

Key Concepts

Divide and conquer algorithm, Design of divide and conquer algorithm, Minmax algorithm, Binary search, Quick sort

Discussion

4.3.1 Divide and Conquer Algorithm

The divide and conquer is an algorithmic pattern. In algorithmic methods, the design takes a dispute on a considerable input, breaks the information into minor pieces, decides the problem on each small amount, and then merges the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer strategy.

The divide and conquer technique has the following strategy. The method divides the problem instance into two or more minor examples of the same problem, solves the more minor models recursively, and assembles the solutions to explain the original illustration. The recursion stops when an instance is reached that is too small to divide. When dividing the model, one can either use whatever division comes most readily to hand or invest time in dividing carefully to simplify the assembly.

Generally, divide-and-conquer algorithms have three parts.

1. **Divide:** Break the problem into several sub-problems that are more minor instances of the same problem.

2. **Conquer:** Solve each of the subproblems recursively. If a subproblem is small enough or trivial, solve it directly.

3. **Combine:** Combine the solutions of the subproblems to form a solution to the original problem

Figure 4.3.1 is the graphical representation of the divide and conquer algorithm. This technique is used to design algorithms that solve problems so that the problem is broken down into one or more minor instances of the same problem, and each smaller model is solved recursively. These more minor instances of the bigger problem are called sub-problems. The method keeps on breaking the sub-problems to a level at which one can solve them directly. Finally, all the solutions to these sub-problems are combined to get the answer for the main problem. Depending upon the size of the sub-problem, there are two cases: recursive case and base case. The sub-problem size is large enough in the recursive case, and they have to be solved recursively. However, in the base case, the sub-problem size is small enough to be solved directly. Many computer algorithms are based on the divide-and-conquer approach. They are maximum and minimum problems, binary search, sorting (merge sort, quick sort), and tower of Hanoi.

There are two fundamental concepts in the divide and conquer strategy: relational

formula and stopping condition. The relational formula is the formula that is generated from the given technique. After the generation of the formula, apply the divide and conquer strategy, i.e. break the problem recursively and solve the broken sub-problems. When breaking the problem using the divide and conquer strategy, the method needs to know how much time it takes to apply the divide and conquer technique. The condition where the process needs to stop the recursion steps of the divide and conquer strategy is called a stopping condition.

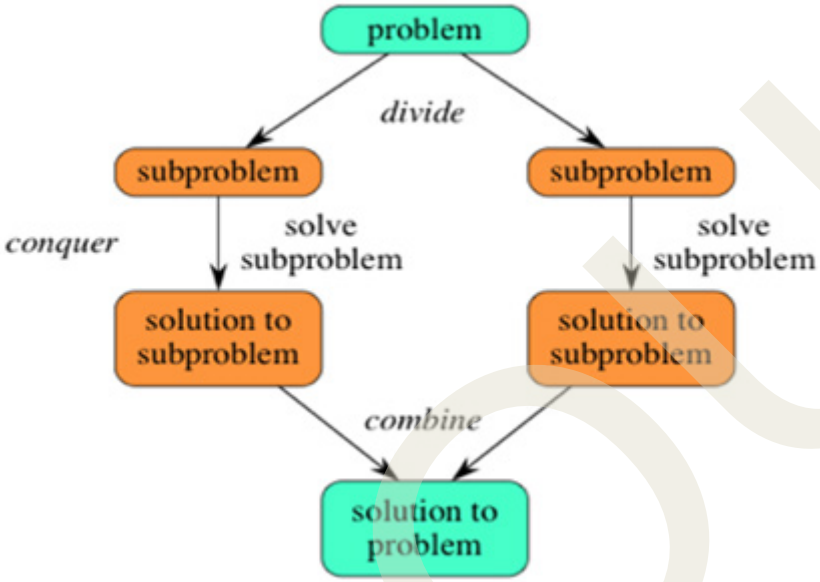


Figure 4.3.1 Divide and Conquer

4.3.1.1 Advantages of Divide and Conquer Algorithm

- The divide and conquer paradigm allows us to solve complex and often impossible-looking problems, such as the Tower of Hanoi, a mathematical game or puzzle.
- The divide and conquer method reduces the degree of difficulty since it divides the problem into subproblems that are easily solvable and usually run faster than other algorithms.
- The divide and conquer algorithm often plays a part in finding other efficient algorithms, and in fact, it plays the central role in finding the quick sort and merge sort algorithms.
- The divide and conquer method uses memory caches effectively. This is because when the sub-problems become simple enough, they can be solved within a cache without having to access the slower main memory, which saves time and makes the algorithm more efficient.
- The divide and conquer method can even produce more precise computations with rounded arithmetic than iterative methods would.

4.3.1.2 Disadvantages of Divide and Conquer Algorithm

- The divide and conquer algorithm issue is that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.

- The divide and conquer method can sometimes become more complicated than a basic iterative approach, especially in large sub-problems.
- In the divide and conquer method, sometimes, once the problem is broken down into sub-problems, the same sub-problem can occur many times.
- In the divide and conquer method, it cannot often be easier to identify and save the solution to the repeated sub-problem, which is commonly referred to as memorisation.
- The divide and conquer algorithm may require more memory space due to recursion

4.3.2 Examples of Divide and Conquer Algorithms

4.3.2.1 Binary Search

The binary search algorithm is the search technique that works efficiently on sorted lists. However, the searching process works only on a sorted set of elements. The binary search algorithm follows the divide and conquer approach in which the sorting list is divided into two halves, and the item is compared with the middle element of the list. If the matching item is found in the list, then the location of the central element is returned; otherwise, search into either of the halves depending upon the result produced through the match.

When a binary search algorithm is used to perform operations on a sorted set, the number of iterations can permanently be reduced based on the value that is being searched.

Binary Search Algorithm Implementation Steps

Binary search cannot be used to find a list of elements arranged in random order. This searching process is applied only to a sorted list of elements. The binary search process starts by comparing the search element with the middle element in the list. The binary search algorithm is implemented using the following steps.

Step 1: Read the search element from the user.

Step 2: Find the middle element in the sorted list.

Step 3: Compare the search element with the middle element in the sorted list.

Step 4: If both are matched, then display "Given element is found" and terminate the function.

Step 5: If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6: If the search element is smaller than the middle element, repeat steps 2, 3, 4 and 5 for the left sub-list of the middle element.

Step 7: If the search element is larger than the middle element, repeat steps 2, 3, 4 and 5 for the right sub-list of the middle element.

Step 8: Repeat the same process until we find the search element in the list or until the list contains only one element.

Step 9: If that element also doesn't match with the search element, then display "Element is not found in the list" and terminate the function.

Example 1

Consider the following list of elements and the elements to be searched.

Fig 4.3.2 Array of integers

Step1: search element (12) is compared with the middle element (50)

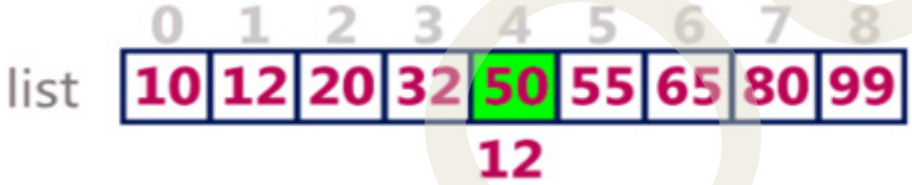


Fig 4.3.3 Array of integers

Both do not match, and element 12 is smaller than 50. So search only in the left sub-list (that is 10, 12, 20 and 32).

Fig 4.3.4 Array of integers

Step2: search element (12) is compared with the middle element (12)

Fig 4.3.5 Array of integers

Both are matching. Therefore, the result is "Element found at index 1"

Example 2

Consider the following list of elements and the element to be searched.

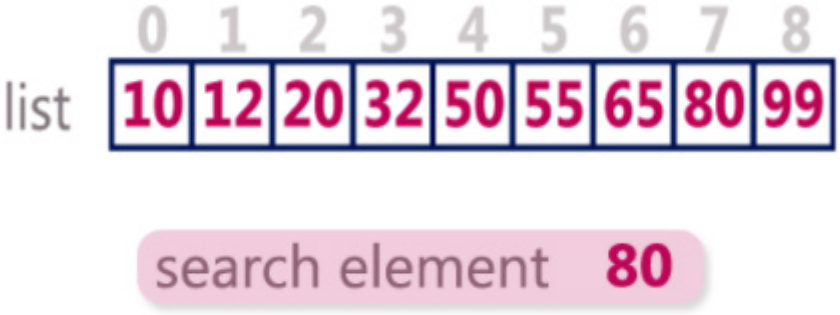


Fig 4.2.6 Array of integers

Step1: search element (80) is compared with the middle element (50)

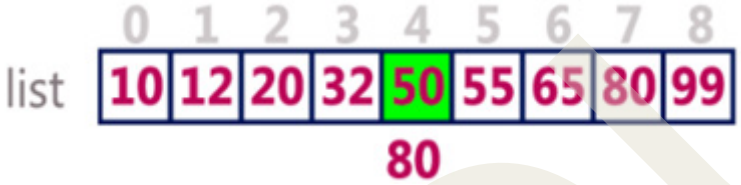


Fig 4.3.7 Array of integers

Both do not match. The element (80) is larger than the element (50). So, search only in the right sub-list (that is, 55, 65, 80 and 99).

Fig 4.3.8 Array of integers

Step 2: The search element (80) is compared with the middle element (65)

Fig 4.3.9 Array of integers

Both do not match. The element 80 is larger than 65. So search only in the right sub-list (that is 80 and 90).

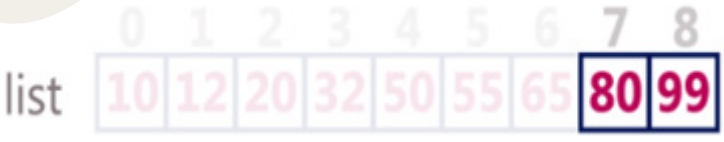


Fig 4.3.10 Array of integers

Step 3: The search element (80) is compared with the middle element (80).

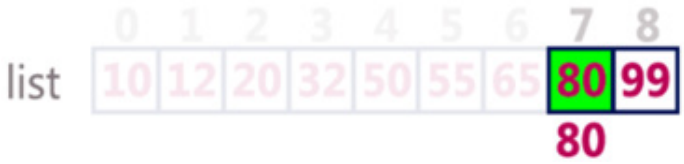


Fig 4.3.11 Array of integers

Both are matching. So the result is "Element is found at index 7".

The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

4.3.2.2 Quick Sort Algorithm

British computer scientist Tony Hoare developed the algorithm in 1959. The name "Quicksort" comes because quick sort can sort a list of data elements significantly faster (twice or thrice faster) than any of the standard sorting algorithms. It is one of the most efficient sorting algorithms. It is based on the splitting of an array (partition) into smaller ones and swapping (exchange) based on the comparison with the "pivot" element selected. Due to this, quick sort is also called the "Partition Exchange" sort.

The quicksort is a divide-and-conquer algorithm. Like all the divide and conquer algorithms, it first divides a larger array into smaller sub-arrays and recursively sorts the sub-arrays. Three steps are involved in the whole sorting process of the quick sort algorithm they are;

1. Pivot selection: Pick an element from the array (usually the leftmost or the rightmost elements of the portion).
2. Partitioning: Reorder the array such that all elements with a value less than the pivot come before the pivot. In contrast, all elements with values greater than the pivot come after it. The equal values can go either way after this partitioning. The pivot is in its final position.
3. Recur: Recursively apply the above steps to the sub-array of elements with smaller values in the pivot and separately to the sub-array of elements with higher values than the pivot.

Quick Sort Algorithm Implementation Steps

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it uses a divide-and-conquer strategy. The heart of quick sort is the partition algorithm. The partition of the list is performed based on the element called pivot. The pivot element is one of the elements in the list. The list is divided into two partitions such that "all elements to the left of the pivot are smaller than the pivot and all elements to the right of the pivot are greater than or equal to the pivot".

Consider an array arr[5]= {10,80,30,90,40}. Take the last element as a pivot.

Fig.4.3.12 Initial array

Partition Function:

1. The pivot is chosen as the last element in the array (pivot = arr[high]).
2. i is initialised to low - 1.
3. The loop runs from low to high - 1. If the current element is smaller than or equal to the pivot, increment i and swap arr[i] with arr[j].
4. After the loop, swap the pivot element with the element at i + 1 to place the pivot in its correct position.
5. Return the partition index i + 1.

Compare 10 with the pivot. Since it is less than the pivot, position it accordingly.

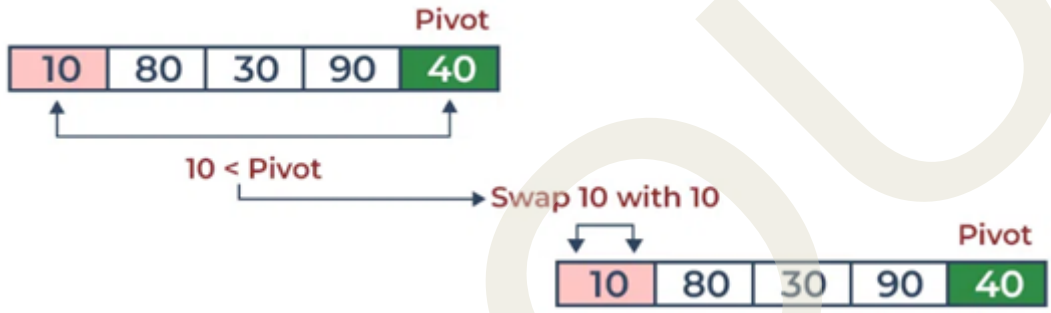


Fig.4.3.13 Array after first comparison

Compare 80 with the pivot. Since it is greater than the pivot, place it accordingly.

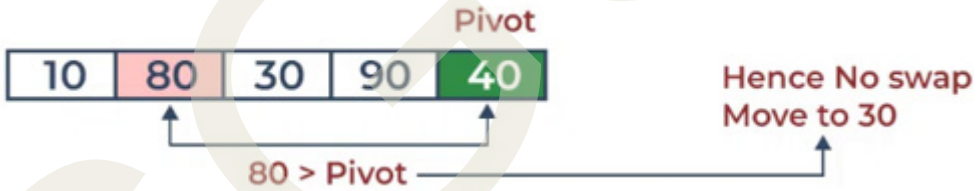


Fig.4.3.14 Array after second comparison

Compare 30 with pivot. It is less than a pivot, so arrange it accordingly.

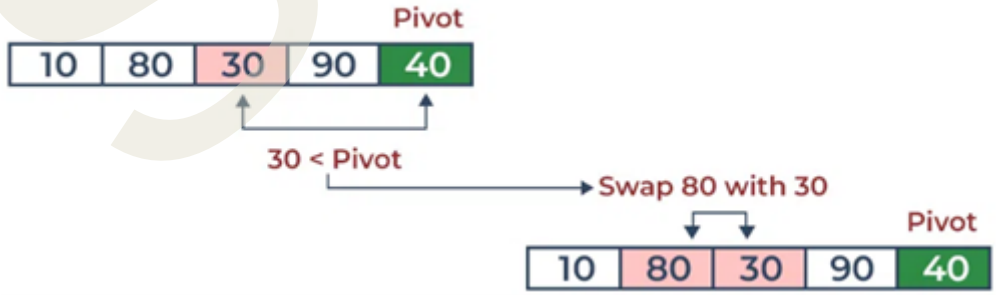


Fig.4.3.15 Array after 3rd comparison

Compare 90 with the pivot. It is greater than the pivot.

Fig.4.3.16 Array after 4th comparison

Initial partition on the main array

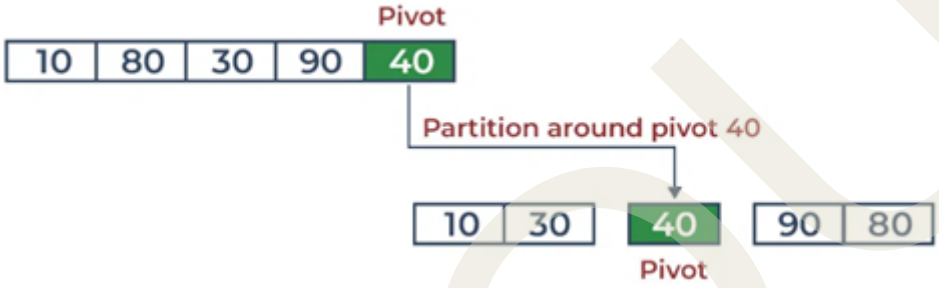


Fig.4.3.17 Array after first partitioning

Now, the array is divided into two partitions where all the elements to the left of 40 are less than 40, and the elements to the right of 40 are greater than 40. Continuing the process by applying the partition algorithm on both the partitions, we will get the array sorted.

4.3.3 Backtracking

Backtracking is a method used for solving problems by exploring all possible solutions incrementally and abandoning those that fail to meet the criteria for a valid solution. It is particularly useful for tackling combinatorial challenges, such as puzzles, mazes, and problems requiring constraint satisfaction, by systematically searching through a large set of potential solutions. The technique involves making a series of choices, each one extending the current solution path. If a chosen path proves invalid, the method backtracks to the last valid point and tries a different path. This systematic approach ensures that all potential solutions are considered, making backtracking a robust tool for finding feasible solutions to complex problems.

4.3.3.1 Mini-max Algorithm

Mini-max is a decision-making algorithm typically used in a turn-based, two-player game. The goal of the algorithm is to find the optimal next move. In the algorithm, one player is called the maximiser, and the other player is a minimiser. Both these players play the game as one tries to get the highest score possible or the maximum benefit while the opponent tries to get the lowest score or the minimum help. Every game has an evaluation score assigned to it to select the maximised value, and the minimiser will

determine the minimised value with counter moves. If the maximiser has the upper hand, then the board score will be a positive value, and if the minimiser has the upper hand, then the board score will be a negative value. Therefore, for one player to win, the other has to lose.

Definition: The Mini-max algorithm is a recursive or backtracking algorithm used in decision-making and game theory. It provides an optimal move for the player, assuming that the opponent is also playing optimally.

The maximizer will try to get the maximum possible score, and the minimizer will try to get the minimum possible score. Then, at the terminal node, the terminal values are given, finally, compare those values and backtrack the tree until the initial state occurs.

4.3.3.2 The Mini-Max algorithm for solving the two-player game tree

The below-given steps are involved in the mini-max algorithm for solving the two-player game tree.

Step 1: In the first step, the algorithm generates the entire game tree and applies the utility function to get the utility values for the terminal states. In the below tree diagram,
Let us take A as the initial state of the tree.

Suppose maximizer takes the first turn, which has worst-case initial value = - infinity, and minimizer will take next turn which has worst-case initial value = +infinity.

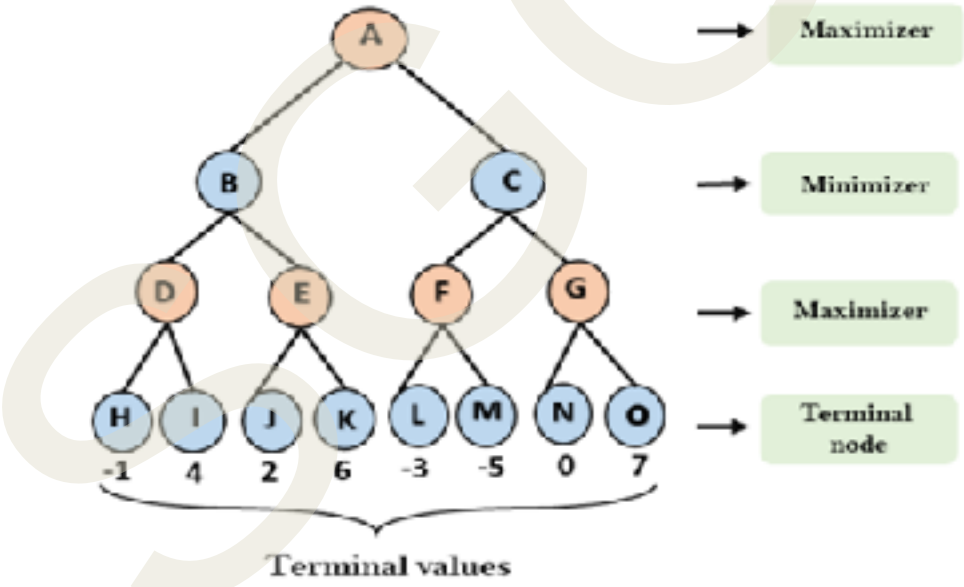


Fig 4.3.18 Initial Stage in Mini-Max Algorithm

Step 2: Find the utility value for the maximizer; its initial value is -∞, so compare each value in the terminal state with an initial value of the maximizer and determine the higher nodes values. It will find the maximum among them all. For example, from Figure 4.3.4, the node value of D, E, F, and G will be 4, 6, -3, and 7, respectively.

- For node D max(-1, -∞) => max(-1,4)= 4

- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7

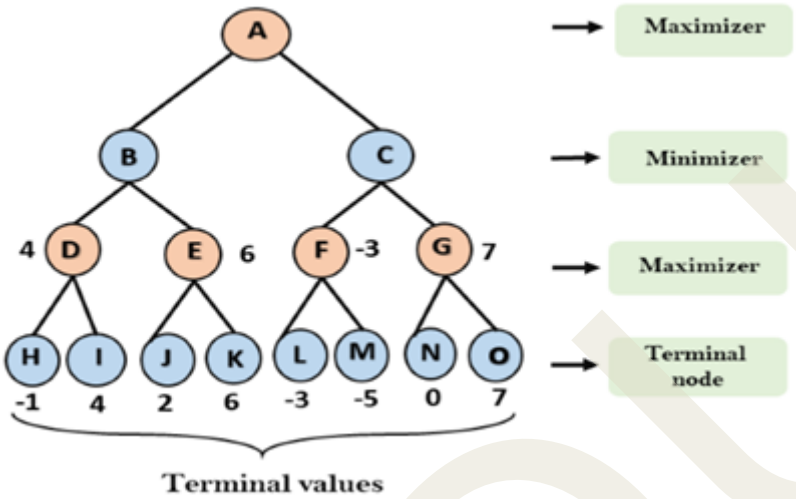


Fig 4.3.19 Second Stage in Mini-Max Algorithm

Step 3: In the third step, it is a turn for the minimiser, so it will compare all node values with +∞ and find the 3rd layer node values.

- For node B = min(4,6) = 4
- For node C = min (-3, 7) = -3

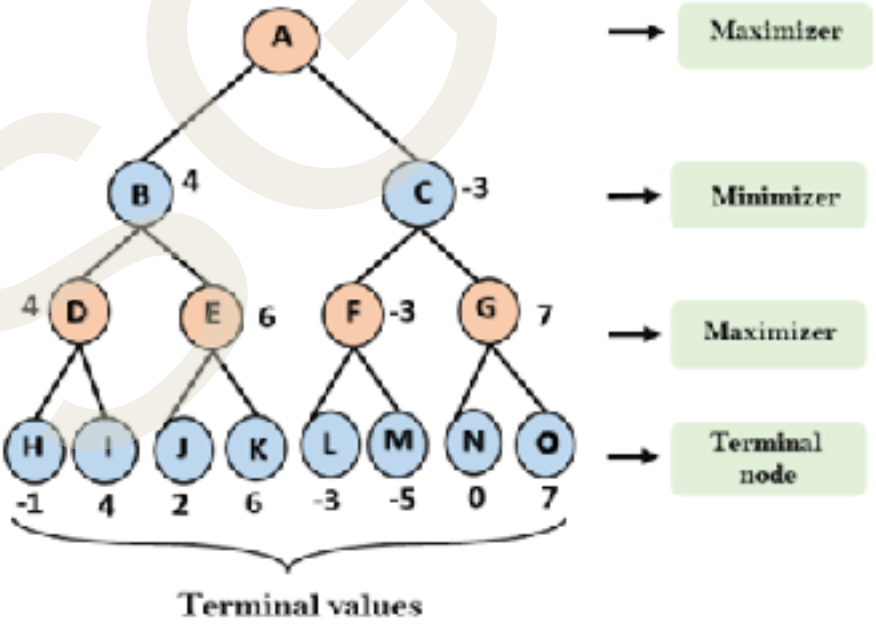


Fig 4.3.20 Third Stage in Mini-Max Algorithm

Step 4: Now, it is a turn for the maximiser. Figure 4.3.6 represents the maximum of all

node values, and the method finds the maximum value for the root node. In this game tree, there are only four layers. Hence, reach the root node immediately, but there will be more than four layers in actual games.

For node A max(4, -3)= 4

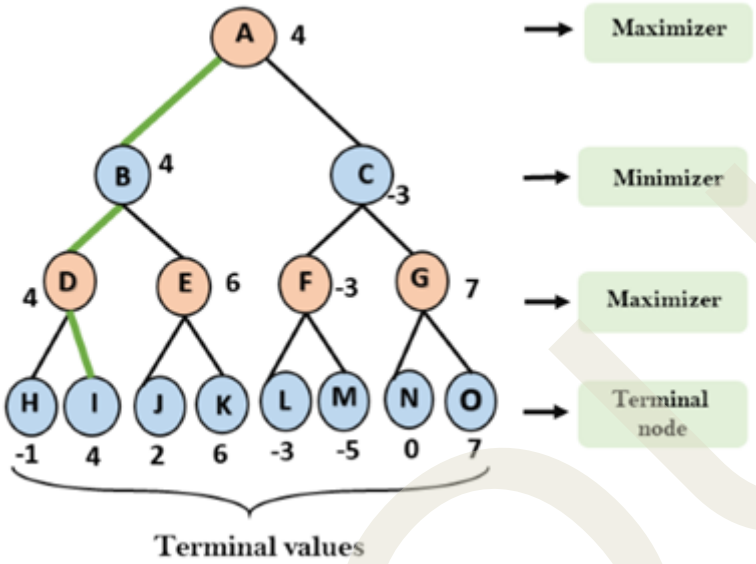


Fig 4.3.21 Fourth Stage in Mini-Max Algorithm

That was the complete workflow of the mini max two-player game algorithm. The main drawback of the mini max algorithm is that it gets slow for complex games such as Chess, go, etc.

Recap

- A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same type until these become simple enough to be solved directly.
- Binary search is a fast search algorithm, and it works on the principle of divide and conquer.
- Binary search is commonly known as a half-interval search or a logarithmic search.
- Quick sort is a highly efficient sorting algorithm based on the partitioning of an array of data into smaller arrays.
- The minimax algorithm is recursive, and it is used in decision-making and game theory.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.

Objective Type Questions

1. Which approach is similar to dynamic programming?
2. Which approach is used in merge sort, quick sort and binary search?
3. What are the recursive steps in the divide and conquer algorithm?
4. Quick sort follows the Divide-and-Conquer strategy. Is the given statement true?
5. Which search is similar to mini-max search?
6. Which kind of sorting is partition and exchange sort?
7. What is rearranging pairs of elements, which are out of order, until no such pairs?
8. Which problem cannot use binary search?
9. What is the method used by card sorters?

Answers to Objective Type Questions

1. Divide and conquer algorithm.
2. Divide and conquer algorithm.
3. Divide, Conquer and Combine
4. True.
5. Depth-first search.
6. Quick sort
7. Sorting; specifically bubble sort
8. Unsorted list
9. Insertion sort

Assignments

1. Explain what binary search is?
2. What is the backtracking process?
3. Describe any two advantages of a binary search algorithm?
4. Which type of lists or data sets are binary searching algorithms used for?
5. Explain the phases of divide and conquer.
6. Explain how quicksort works?
7. Given the following list of numbers [14, 17, 13, 15, 19, 10, 3, 16, 9, 12], what is the answer that shows the list's contents after the second partitioning according to the quick sort algorithm?
8. What are the operations performed with the priority queue?
9. Write a few applications for the priority queue.
10. Mention the overflow and underflow conditions of the circular queue.
11. What is the benefit of the circular queue over the linear queue?
12. What is the application of circular queue?
13. Comparing and finding out the common difference between a queue and a circular queue.
14. Write a C program to implement a queue using an array.
15. Write a C program to implement a circular queue using arrays?

Suggested Reading

1. Data Structures and Algorithms, A. Aho, J. Hopcroft, and J. Ullman, Computer Science and Information Processing Addison-Wesley, Reading, Massachusetts, 1st edition, (Jan 11, 1983)
2. Kruse, Robert, and C. L. Tondo. Data structures and program design in C. Pearson Education India, 2007.
3. Tenenbaum, Aaron M. Data structures using C. Pearson Education India, 1990.
4. Horowitz, Ellis, Sartaj Sahni, and Susan Anderson-Freed. Fundamentals of data structures. Vol. 1982. Potomac, MD: Computer Science Press, 1976.

UNIT 4

Minimum Cost Spanning Trees

Learning Outcomes

By the completion of this unit, the learner will be able to:

- make students aware of the need for minimum cost-spanning trees
- describe the construction of minimum-cost spanning trees
- introduce Prim's algorithm
- familiarise Kruskal's algorithm

Prerequisites

Tree and Graph

Both the tree and graph are non-linear data structures. In a non-linear data structure, the data is stored in a distributed manner. That is, the data is stored in non-sequential form. So, there is no previous or next element. Elements in the non-linear data structure do not form a sequence. There is no unique predecessor or unique successor. In the tree data structure, there is a hierarchical relationship between parents and children. That is one parent and many children. A tree is a special type of graph.

In a graph, the relationship is less restricted. That is the relationship between many parents and many children. Graphs are data structures that have wide-ranging applications in real life, like airlines, analysis of electrical circuits, finding the shortest routes, flow charts of a program, etc. Many real-world problems can be modelled using a graph. Graphs can be used to represent any collection of objects having some kind of pairwise relationship.

Traversal, Shortest Path

Traversal is a searching technique used in graphs. To search for a particular vertex in a given graph, we use traversal. So, the main goal of graph traversal is to find all vertices or nodes that are reachable from a given set of nodes. The traversal also decides the order of vertices visited in a search process. It also finds the edges to be used without creating any loops. We can follow all the edges in an undirected graph. But in a directed graph, we can only follow the out edges. The main graph traversal techniques are Breadth First Search (BFS) and Depth First Search (DFS).

To find the shortest path between two vertices in a graph, first, we have to see all the possible paths that exist between the vertices. Calculate the length of each path by adding the corresponding edge weights included in each path. Then, select a path with the shortest length. The starting vertex is known as the source vertex, and the ending vertex is known as the destination vertex. A path from source vertex s to the destination vertex t is the shortest path from s to t if there is no path from s to t with lower weights.

Key Concepts

Concept of Spanning Trees, Minimum Spanning Tree (MST), Prim's Algorithm, Kruskal's Algorithm

Discussion

4.4.1 Concepts of Spanning Trees

Suppose you are a cable TV network distributor, and you have to make new connections in a particular residential area. Let's assume that the area includes 10 houses. You are going to give connections to 10 houses in that area. Fig. 4.4.1 shows the map of that residential area with different routes. With the help of this map, we can easily get an idea about how to connect these 10 houses in various ways. For this, let us assume each house is a node. If a connecting path between 2 houses exists, then we can consider it as an edge between those 2 nodes (houses).

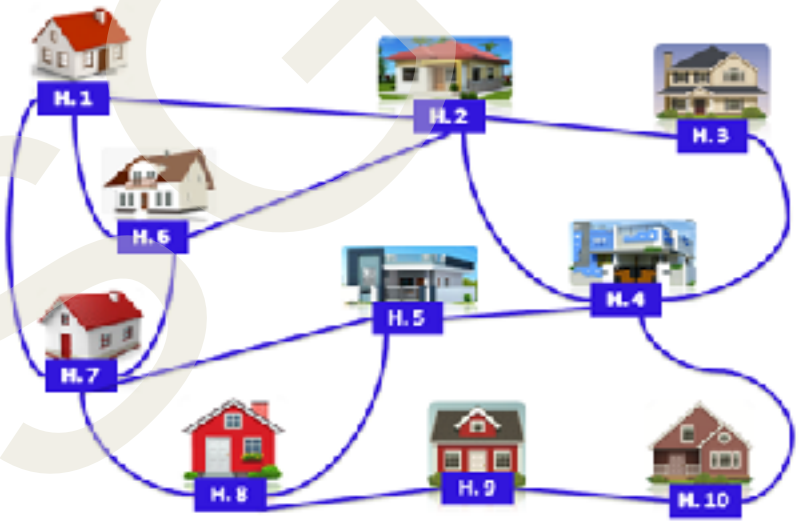


Fig. 4.4.1 A Particular Residential Area Route Map

After constructing all the possible ways, we select one of the best ways (route) to connect all the houses. While creating different routes, we have to consider several things. The main thing is that we must include all the houses in that area. No looping paths are included because it wastes your time and money. So, you have to reject some paths that will create a cycle or loop. Basically, you are constructing a tree that spans

all the nodes of a given graph. Here, the houses and the routes are represented by nodes and graph edges. The tree is acyclic and contains all the nodes of the graph. We can call it a spanning tree of that graph.

A spanning tree of a graph is a subgraph that contains all the vertices of the original graph and forms a tree. If G is a connected graph, its spanning tree includes all the vertices of G and only the edges necessary to connect all these vertices. So the number of edges will be 1 less than the number of nodes. A graph may have many spanning trees. The resultant graph obtained by BFS and DFS is a spanning tree. Spanning tree is basically used to find a minimum path to connect all nodes in a graph.

Properties

- A spanning tree doesn't have cycles, and it cannot be disconnected.
- Every connected and undirected graph has at least one spanning tree.
- A disconnected graph doesn't have any spanning tree as it cannot be spanned to all its vertices.
- A complete undirected graph can have a maximum n^{n-2} number of spanning trees, where n is the number of nodes.
- Removing one edge from the spanning tree will disconnect the graph.
- Adding one edge to the spanning tree will create a circuit or loop.

If a disconnected graph with n vertices has edges less than n-1, then no spanning tree is possible. The applications of spanning trees include civil network planning (water supply, telephone lines, electric lines), cluster analysis, computer network routing protocols, etc. Fig. 4.4.2 shows a connected graph G and its possible spanning trees. Here, we can see that the total number of edges in each spanning tree is 1 less than the number of nodes. (Number of nodes = 4, Number of edges in each spanning tree = 3).

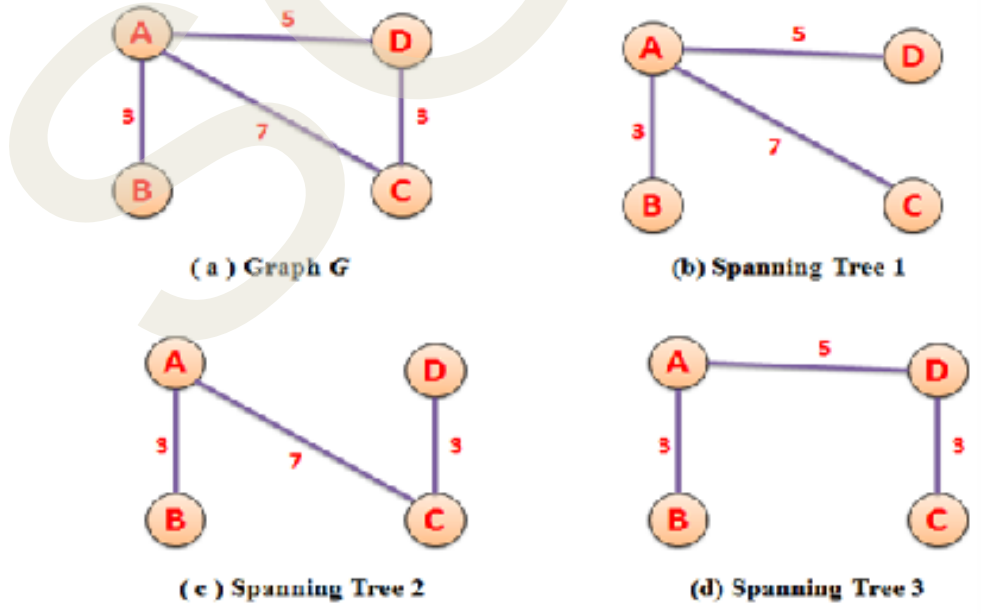


Fig. 4.4.2 Spanning Trees.

4.4.2 Minimum Spanning Tree(MST)

Consider the previous example of cable TV network distribution. We have to consider a possible network of routes that minimise the overall cost. That means our task is to set up lines in such a way that all the houses are connected and the cost of setting up the whole connection is minimal.

Fig. 4.4.3 Cable TV Network example graph and its minimum spanning tree

Fig. 4.4.3 shows the graph representation of this problem. Here, each node represents a house, and each weighted edge represents the cost of setting up the connection between the corresponding two houses. For example, house 1 and house 2 are connected with an edge weighting 10. This means that to set up a connection line between house 1 and house 2, we need a total cost of 10. So, our aim is to reduce the total cost of the entire connection. In Fig. 4.4.3, we can see that all the houses are connected, and the cost of setting up the whole connection is minimised. That is, the sum of all the edges is minimum (here, the minimum cost is 103). We can call it a minimum-cost spanning tree or simply a minimum-spanning tree (MST).

A minimum spanning tree is a spanning tree of a connected, weighted graph G that has the smallest possible sum of edge weights. In other words, it is a tree that includes all vertices of G while minimising the total weight of its edges. The cost of a spanning tree is the sum of the costs of the edges in the tree.

Fig. 4.4.4(a) shows another example of a weighted graph with 4 vertices and 4 edges. We can see all three possible spanning trees of the given graph in Fig. 4.4.4(a). Fig. 4.4.4(b) shows the first spanning tree with a cost of 15, and Fig. 4.4.4(c) shows the second spanning tree with a cost of 13. Fig. 4.4.4(d) is the third spanning tree with a cost of 11. Fig. 4.4.4(d) shows the minimum spanning tree of the given graph G because the sum of the weights of edges in 4.4.4(d) is 11. The sum of the weights of edges in Fig 4.4.4(b) is 15, and in Fig 4.4.4(c) is 13. So, the minimum spanning tree is Fig 4.4.4(d).

A typical application for minimum cost spanning trees occurs in the design of telecommunications networks. The vertices of a graph represent cities and the edges of possible communication links between the cities. The cost associated with an edge represents the cost of selecting that link for the network. A minimum cost spanning

tree represents a communications network that connects all the cities at minimal cost. Minimum spanning trees are also useful in finding the paths in the map. For designing networks like water supply networks and electrical grids, we can use the idea of the minimum spanning tree.

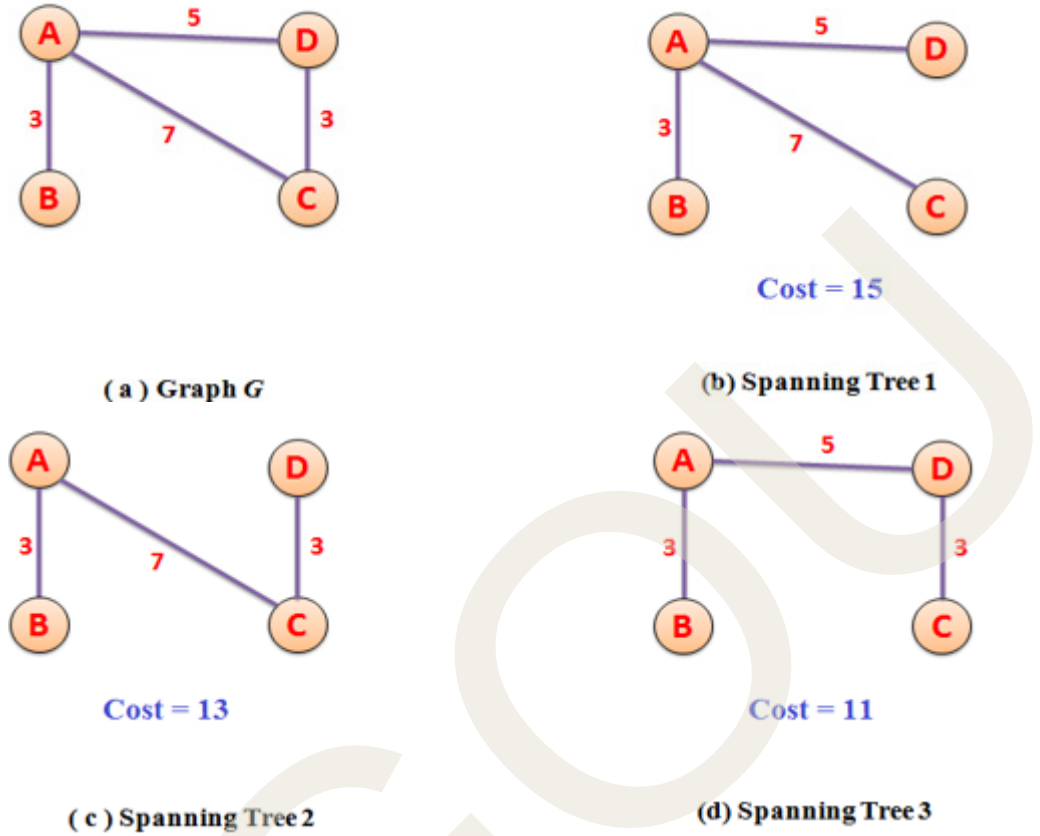


Fig. 4.4.4 Minimum Spanning Tree Example

4.4.3 Prim's Algorithm

Prim's algorithm is used to find the minimum spanning tree of a given graph. We can apply this algorithm to a graph that must be weighted, connected, and undirected. This algorithm finds a subset of the edges that forms a tree that includes every node. Here, the total weight of all the edges of the tree is minimised. In Prim's algorithm, the minimum spanning tree grows naturally, starting from an arbitrary root. At each stage, we add a new branch to the tree already constructed, and the algorithm stops when all the nodes have been reached.

Algorithm

Step 1: Randomly choose any node as the starting node and include that node in the spanning tree.

Step 2: Find all the edges that connect the spanning tree to the new nodes and insert them into the incident edge set.

Step 3: Find the least-weight edge among the edges in the given incident edge set.

Step 4: Check whether including that edge in the existing spanning tree forms a cycle or not.

Step 5: If including that edge creates no cycle, then add that edge to the spanning tree. Otherwise, reject that edge from the incident edge set and go to step 3.

Step 6: Repeat steps 2 to 5 until all the nodes are included in the spanning tree and the minimum spanning tree is obtained.

Example

Consider an example graph shown in Fig. 4.4.5. Now we can see how Prim's algorithm works on the given graph. The given graph contains 7 nodes and 12 weighted edges.

Step 1:

Choose any node from the graph randomly. Here, we choose node "A" as a starting node. Then, it is included in the spanning tree. Fig. 4.4.6 shows the resultant graph and spanning tree.

Step 2:

Now find the edges connecting the new adjacent nodes of A, that is, edges from node A to the new adjacent nodes B and D. Here, 2 edges are there with weights 1 and 5. Insert them into the incident edge set. Now, find the least-weight edge from this set. Here, (A, B) has the least weight edge, and it doesn't form any cycle. So, include the edge (A, B) to the spanning tree. The resultant graph and spanning tree are shown in Fig. 4.4.7.

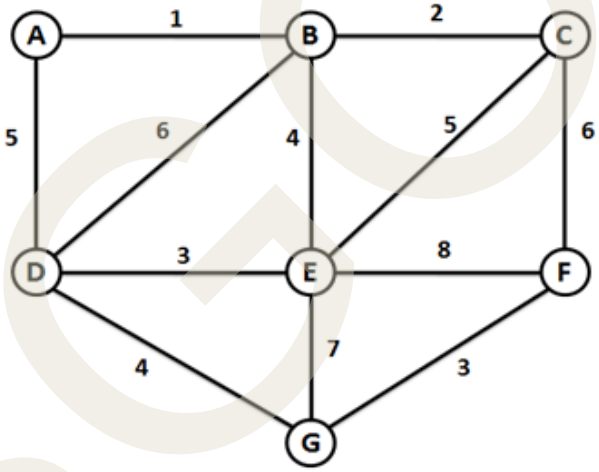


Fig. 4.4.5 Example graph for Prim's algorithm

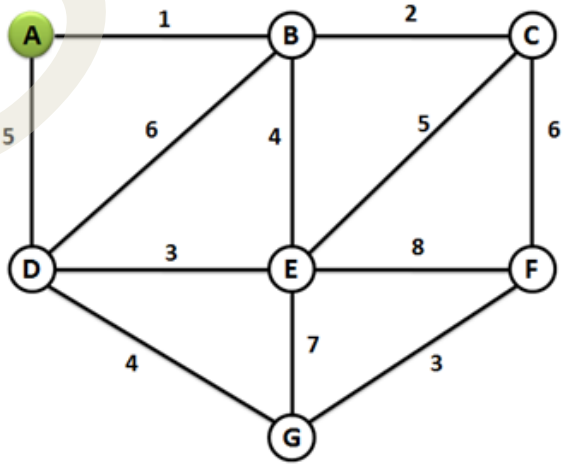


Fig. 4.4.6 Prim's Algorithm – Resultant Graph, Spanning Tree (1)

Step 3:

Now, find the edges connecting the new adjacent nodes of A and B. That is, edges from nodes A and B are moved to the new adjacent nodes C, D, and E. Here, 4 edges are there with weights 5, 6, 4 and 2. That is edges (A, D), (B, D), (B, E), and (B, C). Insert them into the incident edge set. Now, find the least-weight edge from this set. Here (B, C) is the least weight edge, and including this edge to the spanning tree, doesn't form any cycle. So, include the edge (B, C) to the spanning tree. The resultant graph and spanning tree are shown in Fig. 4.4.8.

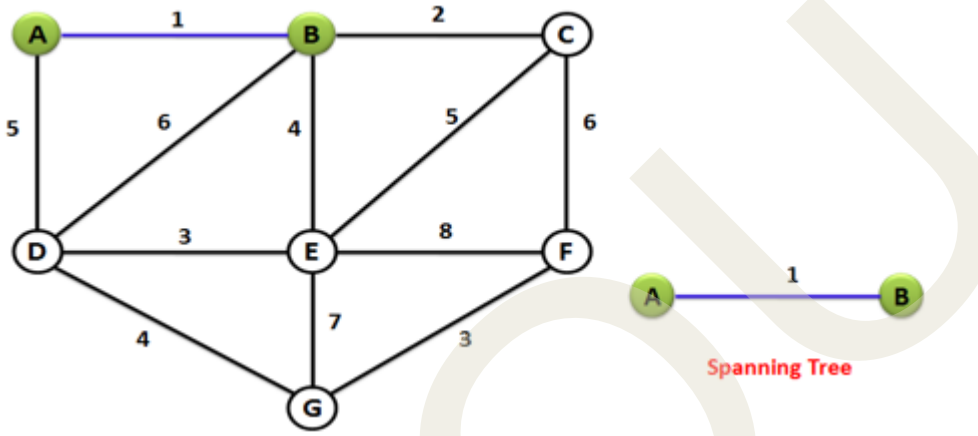


Fig. 4.4.7 Prim's Algorithm – Resultant Graph, Spanning Tree (2)

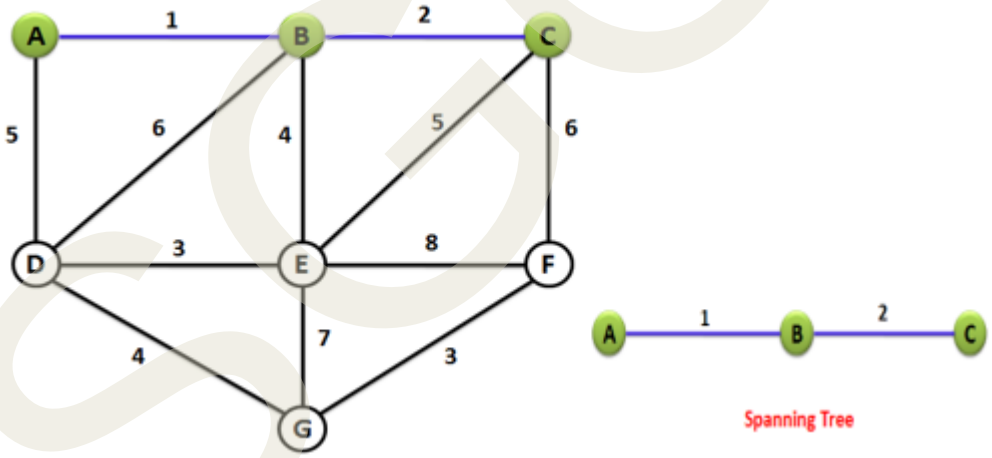


Fig. 4.4.8 Prim's Algorithm – Resultant Graph, Spanning Tree (3)

Step 4:

Now, find the edges connecting the new adjacent nodes of A, B, and C. That is, edges from the nodes A, B, and C to the new adjacent nodes D, E and F. Here, 5 edges have weights 5, 6, 4, 5, and 6. That is, edges (A, D), (B, D), (B, E), (C, E) and (C, F). Insert them into the incident edge set. Now, find the least-weight edge from this set. Here (B, E) is the least weight edge, and including this edge to the spanning tree, doesn't form any cycle. So, include the edge (B, E) to the spanning tree. The resultant graph and spanning tree are shown in Fig. 4.4.9.

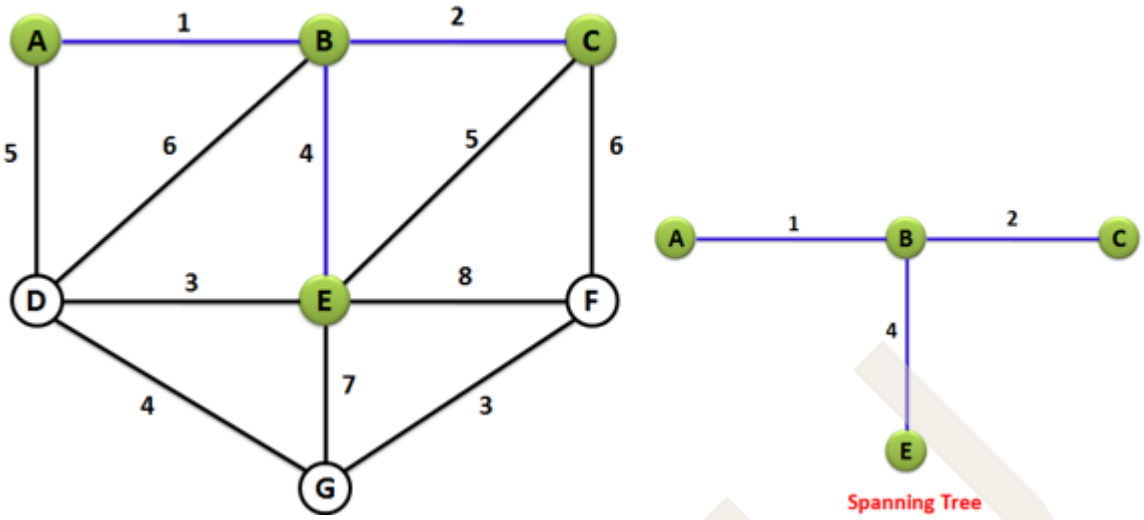


Fig. 4.4.9 Prim's Algorithm – Resultant Graph, Spanning Tree (4)

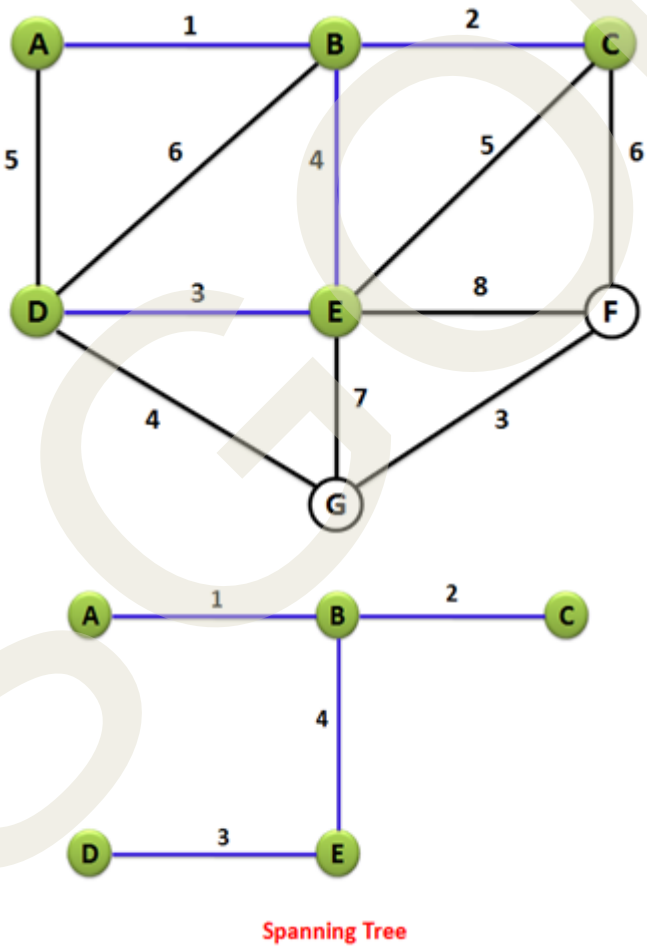


Fig. 4. 4.10 Prim's Algorithm – Resultant Graph, Spanning Tree (5)

Step 5:

Now, find the edges connecting the new adjacent nodes of A, B, C, and E. That is, edges from the nodes A, B, C, and E to the new adjacent nodes D, F, and G. Here, 6 edges

have weights 5, 6, 6, 3, 7, and 8. That is, edges (A, D), (B, D), (C, F), (E, D), (E, G) and (E, F). Insert these 5 edges into the incident edge set. Now, find the least-weight edge from this set. Here (E, D) is the least weight edge, and including this edge to the spanning tree, doesn't form any cycle. So, include the edge (E, D) to the spanning tree. The resultant graph and spanning tree are shown in Fig. 4.4.10.

Step 6:

There are no new adjacent nodes from A and B. So now we have to find the edges connecting the new adjacent nodes of C, E and D. That is, edges from the nodes C, E, and D to the new adjacent nodes G and F. Here, 4 edges are there with weights 6, 8, 7 and 4. That is edges (C, F), (E, F), (E, G) and (D, G). Insert these 4 edges into the incident edge set.

Now, find the least-weight edge from this set. Here (D, G) is the least-weight edge, and including this edge in the spanning tree doesn't form any cycle. So, include the edge (D, G) to the spanning tree. The resultant graph and spanning tree are shown in Fig. 4.4.11.

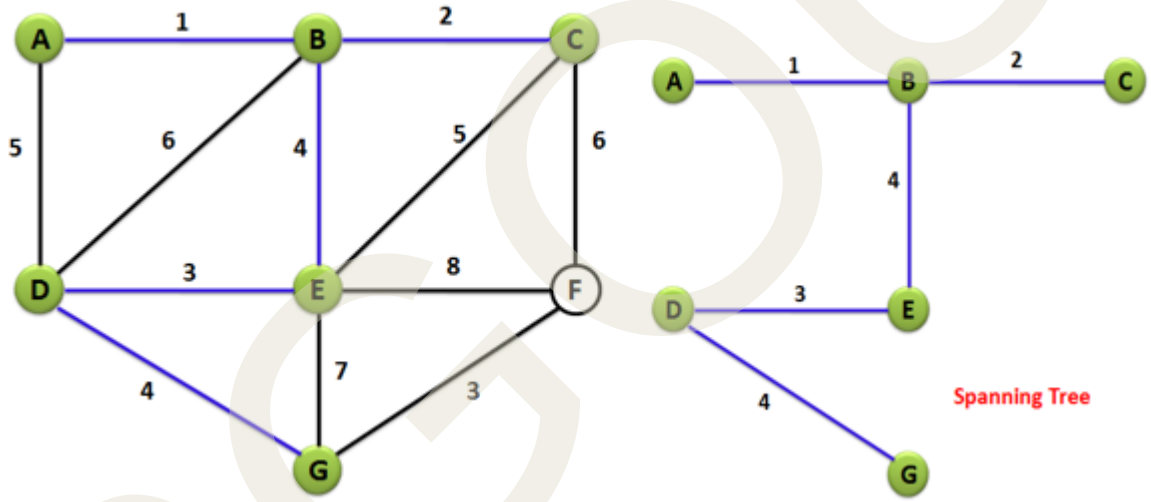


Fig. 4.4.11 Prim's Algorithm – Resultant Graph, Spanning Tree (6)

Step 7:

There are no new adjacent nodes from A, B and D. So now we have to find the edges connecting the new adjacent nodes of C, E, and G. That is, edges from the nodes C, E, and G to the new adjacent node F. Here 3 edges are there with weights 6, 8, and 3. That is edges (C, F), (E, F), and (G, F). Insert these 3 edges into the incident edge set. Now, find the least-weight edge from this set. Here (G, F) is the least weight edge, and including this edge to the spanning tree, doesn't form any cycle. So, include the edge (G, F) to the spanning tree. The resultant graph and its minimum spanning tree are shown in Fig. 4.4.12.

After performing Prim's algorithm on an undirected, weighted and connected graph, the resultant spanning tree formed will be the minimum cost spanning tree. The cost of a spanning tree is the sum of the weights of all the edges present in that spanning tree. Here, 6 edges are in the spanning tree. They are (A, B), (B, C), (B, E), (E, D), (D, G)

and (G, F). The sum of the weights of these six edges is 17. So the minimum cost is 17.

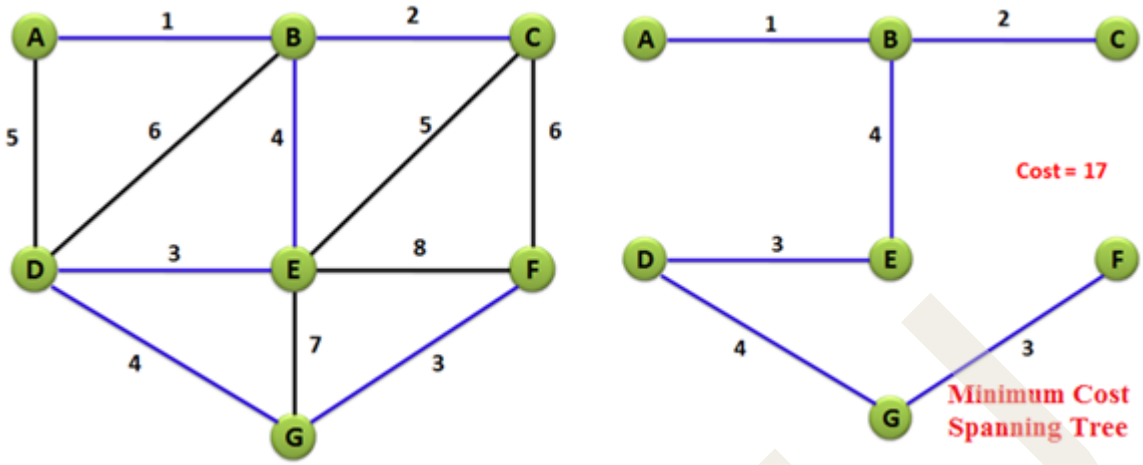


Fig. 4.4.12 Prim's Algorithm – Resultant Graph, Minimum Spanning Tree

4.4.4 Kruskal's Algorithm

Kruskal's algorithm creates a forest of trees. Initially, the forest contains n single node trees and no edges. At each step, we add one edge that joins two trees together. If it creates a cycle, then that edge is not included in the tree. In this method, we first examine all the edges and sort them in the increasing order of their weights. Then, it is stored in a priority queue. That is the first priority that goes to the edge with the least value.

Thus, the edges are examined one by one according to their weights and decide whether the selected edge should be included in the spanning tree or not. If the two nodes of the selected edge belong to the same tree, then we will not include that edge in the spanning tree because the two nodes are in the same tree and are already connected. Adding this edge will create a cycle. So, we reject that edge from the spanning tree. We will insert an edge in the spanning tree only if its nodes are in different trees.

Algorithm

Step 1: Initially, construct a separate tree for each node in a graph.

Step 2: Sort the edges in ascending order according to their weights.

Step 3: Store the sorted edges in a priority queue.

Step 4: Examine all the edges one by one from the priority queue.

Step 4.1 Check whether including the edge will create a cycle or not.

Step 4.2 If it creates no cycle and then adds that edge to the spanning tree,

Otherwise, reject that edge.

Example

Consider an example graph shown in Fig. 4.4.13. Now, we can find the minimum

cost-spanning tree of the following graph using Kruskal's algorithm. The given graph contains 7 nodes and 12 weighted edges. We have to sort the edges in ascending order, according to their weights and store them in a priority queue.

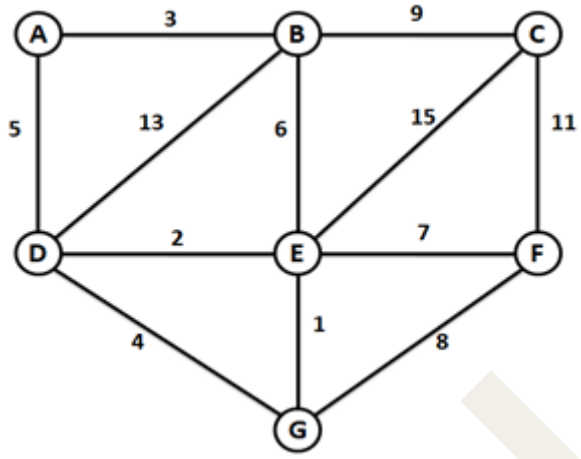
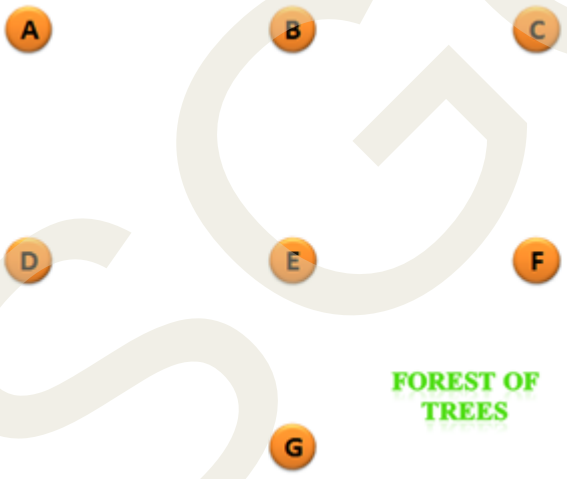


Fig. 4.4.13 Example graph for Kruskal's algorithm

Step 1:

Initially construct a separate tree for each node in a graph. Sort the edges in ascending order according to their weights and store them in a priority queue. Fig. 4.4.14 shows the resultant forest of trees and the priority queue.

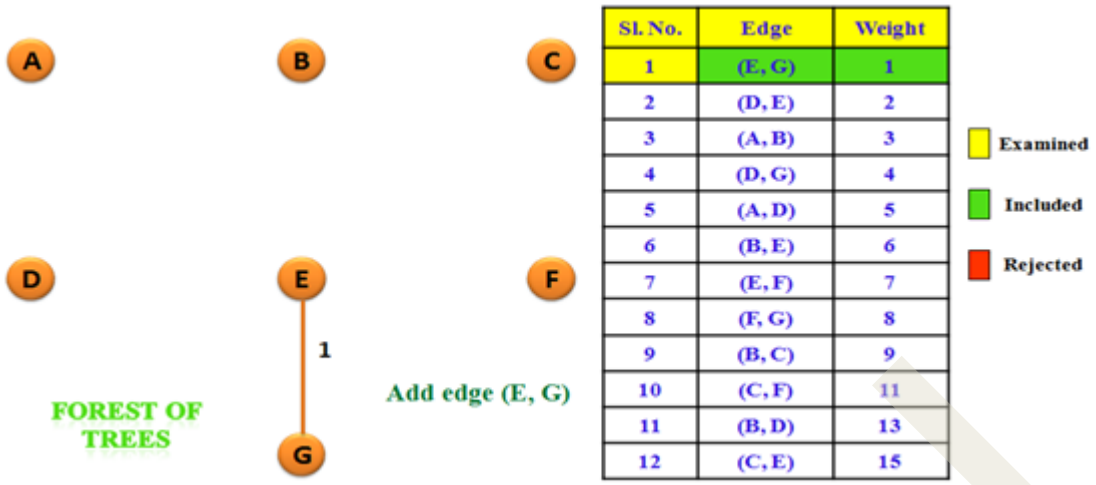


Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig. 4.4.14 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (1)

Step 2:

Now, examine the edges one by one from the priority queue. The least weight edge is (E, G), which is the first entry in the priority queue. Its weight is 1. Check whether including the edge (E, G) in the tree will create a cycle or not. It will not create a cycle. So include (E, G) in the tree. Fig. 4.4.15 shows the resultant forest of trees and the priority queue.

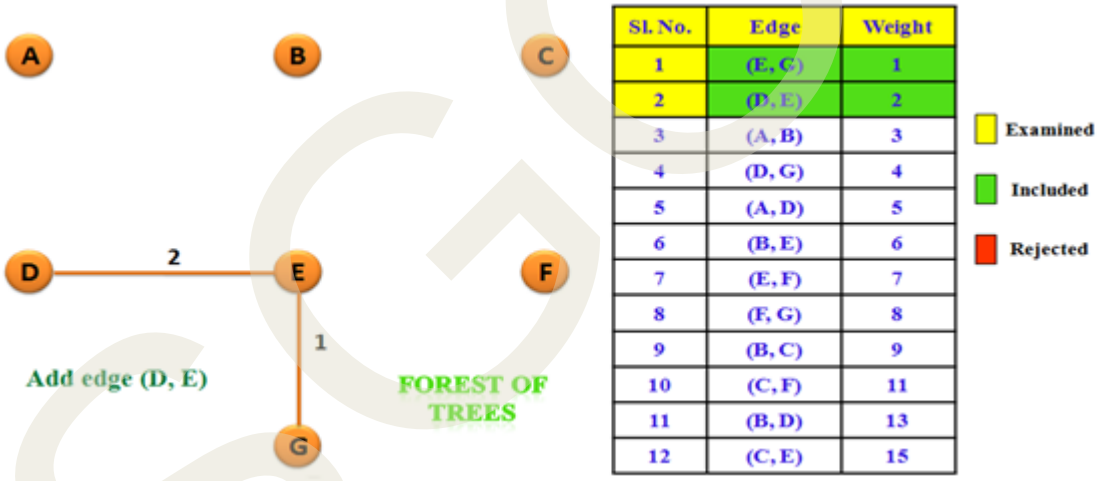


Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig. 4.4.15 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (2)

Step 3:

The next edge is (D, E) with a weight 2. Check whether including the edge (D, E) in the tree will create a cycle or not. It will not create a cycle. So include (D, E) in the tree. Fig. 4.4.16 shows the resultant forest of trees and the priority queue.



Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

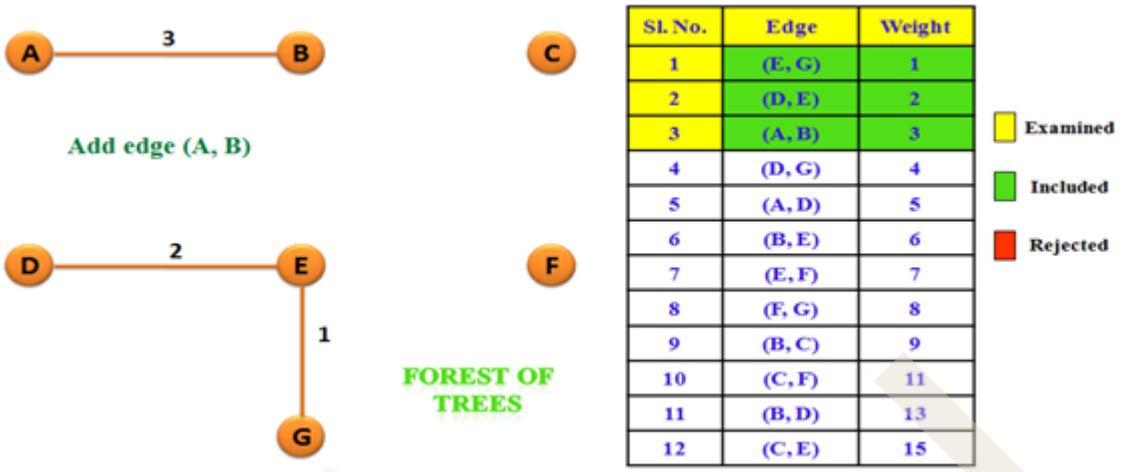
Fig. 4.4.16 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (3)

Step 4:

The next edge is (A, B) with a weight 3. Check whether including the edge (A, B) in the tree will create a cycle or not. It will not create a cycle. So include (A, B) in the tree. Fig. 4.4.17 shows the resultant forest of trees and the priority queue.

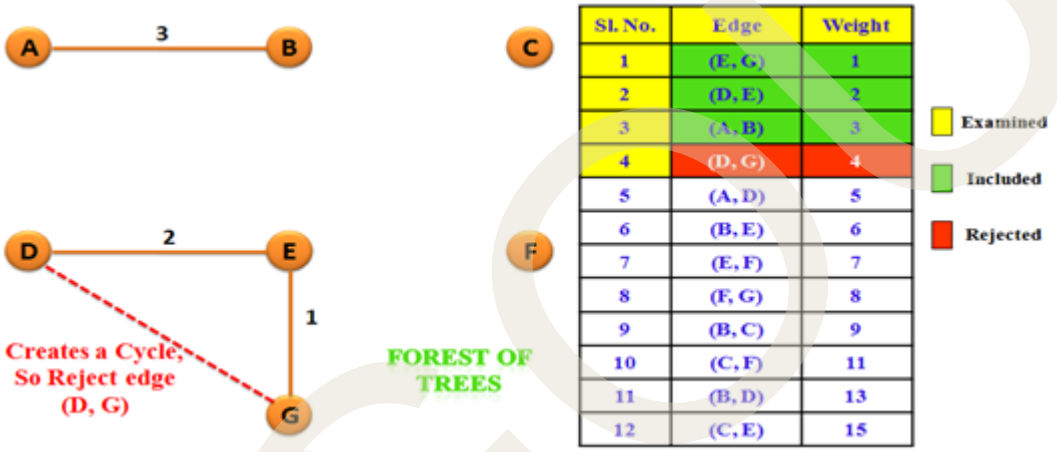
Step 5:

The next edge is (D, G) with a weight 4. Check whether including the edge (D, G) in the tree will create a cycle or not. It will create a cycle. So reject (D, G) from the tree. Fig. 4.4.18 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (D, G). We do not include this edge in the tree.



Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig 4.4.17 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (4)

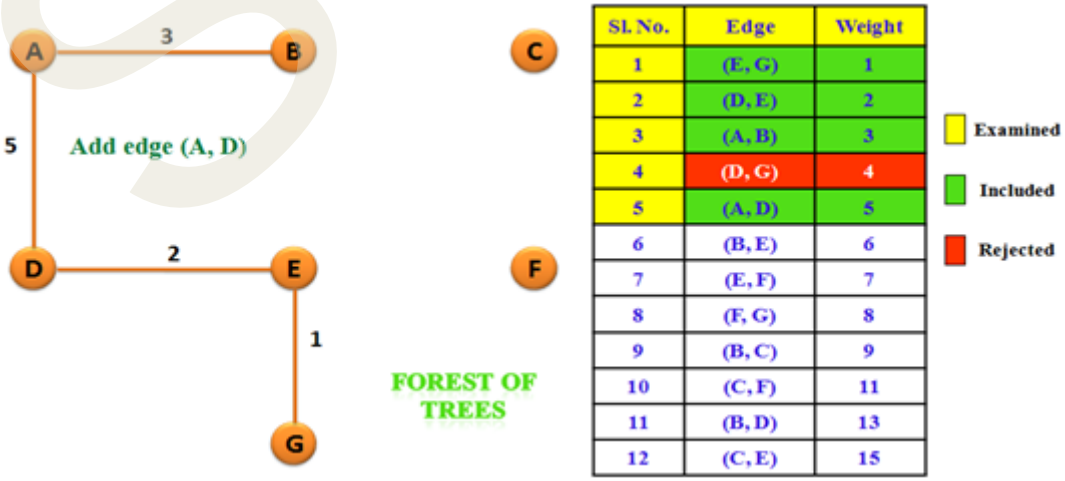


Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig 4.4.18 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (5)

Step 6:

The next edge is (A, D) with a weight 5. Check whether including the edge (A, D) in the tree will create a cycle or not. It will not create a cycle. So include (A, D) in the tree. Fig. 4.4.19 shows the resultant forest of trees and the priority queue.

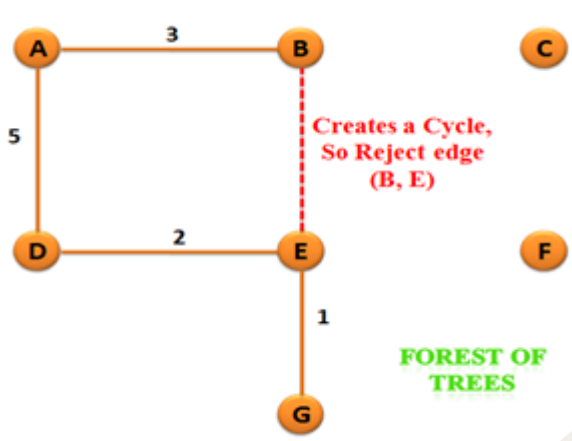


Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig 4.4.19 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (6)

Step 7:

The next edge is (B, E) with a weight 6. Check whether including the edge (B, E) in the tree will create a cycle. It will create a cycle. So, reject the edge (B, E) from the tree. Fig. 4.4.20 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (B, E). We do not include this edge in the tree.



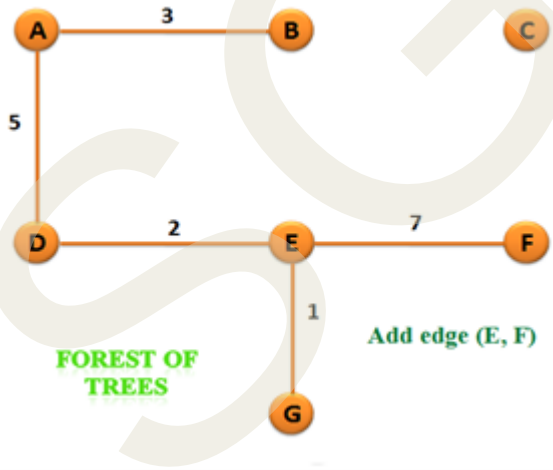
Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Examined
Included
Rejected

Fig. 4.4.20 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (7)

Step 8:

The next edge is (E, F), with a weight of 7. Check whether including the edge (E, F) in the tree will create a cycle or not. It will not create a cycle. So include (E, F) in the tree. Fig. 4.4.21 shows the resultant forest of trees and the priority queue.



Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

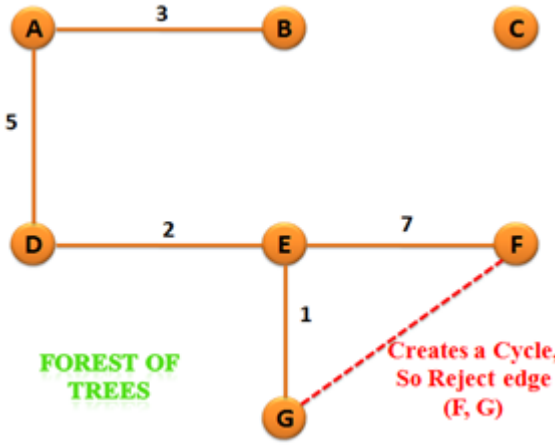
Examined
Included
Rejected

Fig 4.4.21 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (8)

Step 9:

The next edge is (F, G) with a weight 8. Check whether including the edge (F, G) in the tree will create a cycle or not. It will create a cycle. So reject (F, G) from the tree. Fig.

4.4.22 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (F, G). We do not include this edge in the tree.



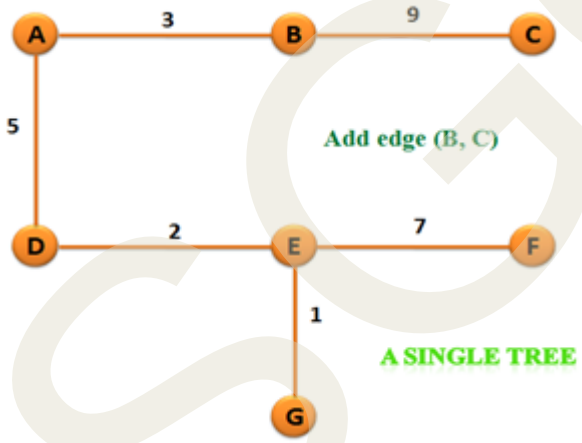
Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Examined
Included
Rejected

Fig 4.4.22 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (9)

Step 10:

The next edge is (B, C) with a weight 9. Check whether including the edge (B, C) in the tree will create a cycle or not. It will not create a cycle. So include (B, C) in the tree. Fig. 4.4.23 shows the resultant forest of trees and the priority queue.



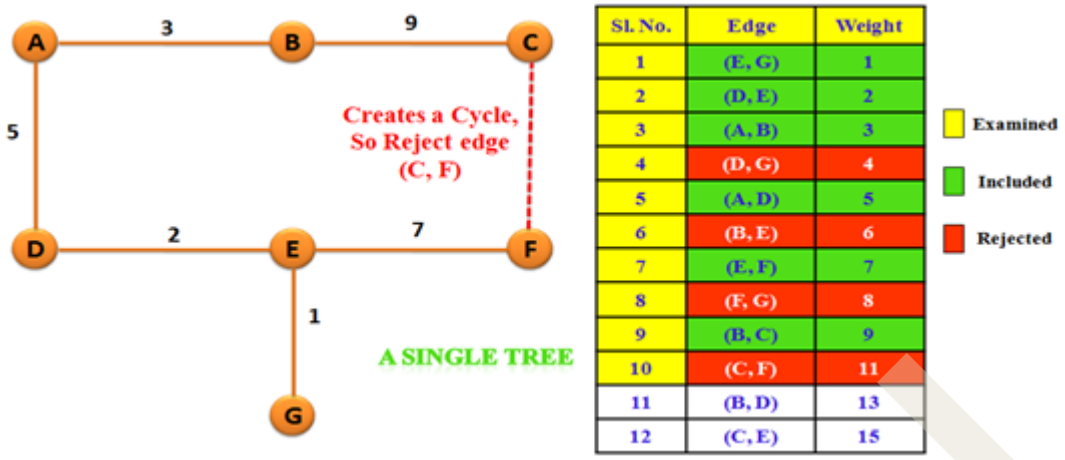
Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Examined
Included
Rejected

Fig 4.4.23 Kruskal's algorithm – Resultant Forest of Trees, Priority Queue (10)

Step 11:

The next edge is (C, F) with a weight 11. Check whether including the edge (C, F) in the tree will create a cycle or not. It will create a cycle. So reject (C, F) from the tree. Fig. 4.4.24 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (C, F). We do not include this edge in the tree.

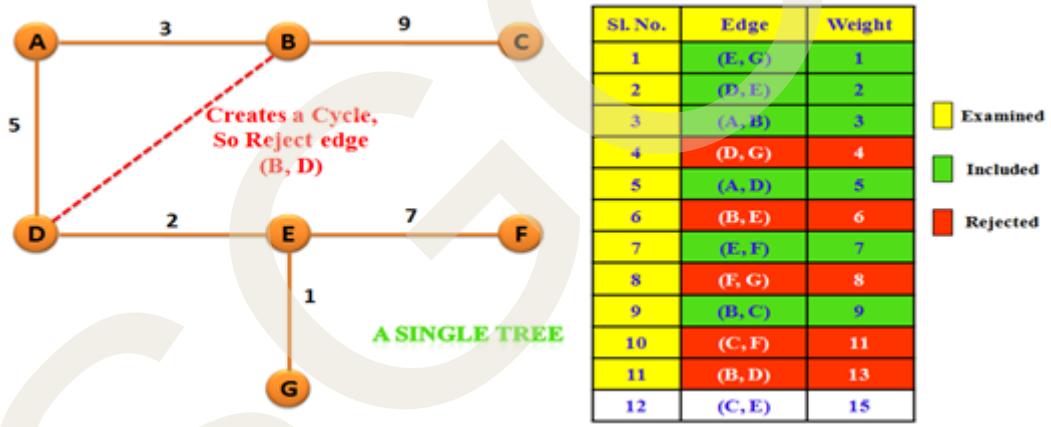


Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig. 4.4.24 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (11)

Step 12:

The next edge is (B, D) with a weight 13. Check whether including the edge (B, D) in the tree will create a cycle or not. It will create a cycle. So reject (B, D) from the tree. Fig. 25 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (B, D). We do not include this edge in the tree.



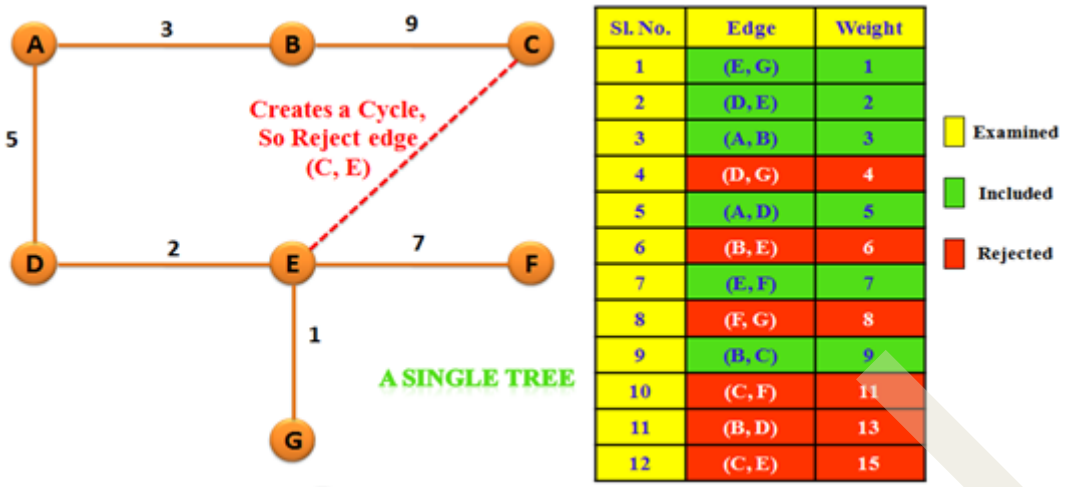
Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig 4.4.25 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (12)

Step 13:

The next edge is (C, E) with a weight 15. Check whether including the edge (C, E) in the tree will create a cycle or not. It will create a cycle. So, reject the edge (C, E) from the tree. Fig. 4.4.26 shows the forest of trees and the resultant priority queue. The red dotted line in the tree shows the rejected edge (C, E). We do not include this edge in the tree.

And the final spanning tree is shown in Fig. 4.4.27. It is the minimum cost spanning tree of the given graph 4. The minimum cost is the sum of the weights of all the edges present in the spanning tree. Here, the sum is 27. So the minimum cost is 27.



Sl. No.	Edge	Weight
1	(E, G)	1
2	(D, E)	2
3	(A, B)	3
4	(D, G)	4
5	(A, D)	5
6	(B, E)	6
7	(E, F)	7
8	(F, G)	8
9	(B, C)	9
10	(C, F)	11
11	(B, D)	13
12	(C, E)	15

Fig 4.4.26 Kruskal's algorithm – Forest of Trees, Resultant Priority Queue (13)

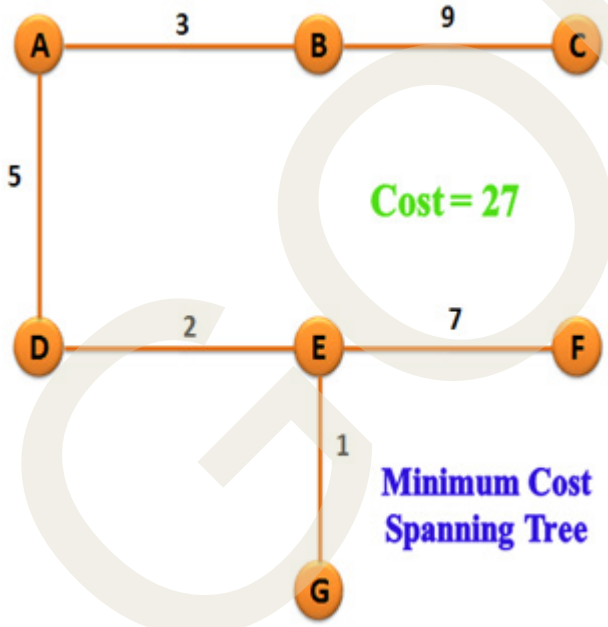


Fig. 4.4.27 Kruskal's Algorithm – Final Minimum Cost Spanning Tree

Recap

- A spanning tree of a graph is a tree that contains all the vertices and some of the edges of the graph.
- The number of edges will be 1 less than the number of nodes.
- A graph may have many spanning trees.
- Spanning tree is basically used to find a minimum path to connect all nodes in a graph.
- A spanning tree doesn't have cycles, and it cannot be disconnected.
- A complete undirected graph can have a maximum n(n-2) number of spanning trees, where n is the number of nodes.
- A spanning tree T that is created such that the sum of the weights of its edges is as small as possible is known as the minimum spanning tree.
- The cost of a spanning tree is the sum of the weights of the edges in the tree.
- Prim's algorithm is used to find the minimum spanning tree of a given graph.
- We can apply Prim's algorithm on a graph that must be weighted, connected and undirected.
- Prim's algorithm finds a subset of the edges that forms a tree that includes every node, and the total weight of all the edges in the tree is minimised.
- In Prim's algorithm, the minimum spanning tree grows naturally, starting from an arbitrary root.
- In Prim's algorithm, at each stage, we add a new branch to the tree already constructed, and the algorithm stops when all the nodes have been reached
- Kruskal's algorithm is used to find the minimum spanning tree of a given graph.
- **Kruskal's algorithm** creates a forest of trees.
    - Initially, the forest contains n single node trees and no edges.
    - In this method, we first examine all the edges and sort them in increasing order according to their weights.
    - The sorted edges are stored in a priority queue.
    - At each step, we add one edge so that it joins two trees together.
    - If an edge creates a cycle, then that edge is not included in the tree.
    - We will insert an edge in the spanning tree only if its nodes are in different trees.

Objective Type Questions

1. A spanning tree of a graph contains all the edges and vertices. Is the given statement true?
2. A graph may have many spanning trees. Is the given statement true?
3. What is the resultant graph obtained by BFS and DFS?
4. What does the spanning tree find in a graph?
5. A spanning tree doesn't have cycles, and it cannot be disconnected. Is the given statement true?
6. Which graph is obtained by removing one edge from the spanning tree will make the graph?
7. Which tree is obtained by adding one edge to the spanning tree?
8. What is the sum of the weights of the edges in the tree?
9. If the cost of a spanning tree is as small as possible, then that spanning tree is known as the minimum spanning tree. Is the given statement true?
10. What does Prim's algorithm find in a given graph?
11. We can apply Prim's algorithm on an undirected graph with no weights. Is the given statement true?
12. Using Prim's algorithm, we can choose any node randomly as a starting node. Is the given statement true?
13. What Prim's algorithm doesn't form in the tree?
14. Which tree contains all the edges that connect the spanning tree to the new nodes?
15. When is an edge added to the spanning tree in Prim's algorithm?
16. At each stage, we add a new branch with the least weight from the given incident edge set to the tree already constructed. Is the given statement true?
17. Prim's algorithm finds a subset of the edges that forms a tree that includes every node. Is the given statement true?
18. What is created by Kruskal's algorithm?
19. What does Kruskal's algorithm find on a given graph?

20. The initial forest formed by Kruskal's algorithm contains n single node trees and no edges. Is this true?

21. If the two nodes of the selected edge belong to the same tree, then we will include that edge in the spanning tree. Is the given statement true?

22. In which order can we insert an edge in the spanning tree?

23. We reject an edge from the spanning tree if it creates a cycle. Is the given statement true?

24. The edges are stored in the priority queue in the ascending order of their weights. Is the given statement true?

Answers to Objective Type Questions

1. False
2. True
3. spanning tree
4. minimum path
5. True
6. disconnected
7. circuit or loop
8. the cost of a spanning tree
9. True
10. minimum spanning tree
11. False
12. True
13. cycles
14. incident edge set
15. including that edge creates no cycle

16. True

17. True

18. forest of trees

19. minimum spanning tree

20. True

21. False

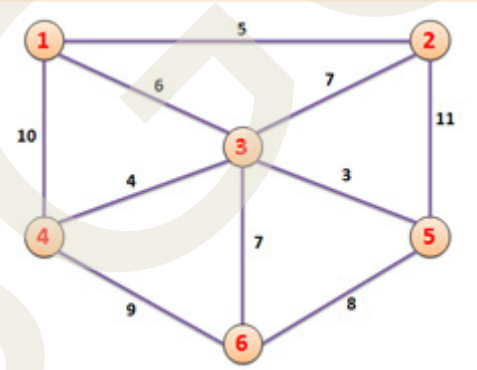
22. different trees

23. True

24. True

Assignments

1. Construct all the possible spanning trees of the following graph. Also, find the Minimum Spanning Tree.



2. Explain spanning trees.

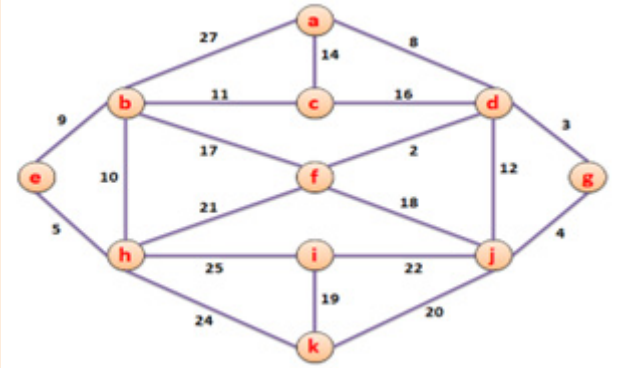
3. What is the minimum cost spanning tree? Explain an example with an algorithm.

4. Explain Prim's algorithm with an example.

5. Explain Kruskal's algorithm with an example.

6. Find the Minimum Spanning Tree of the following graph using
   a. Prim's algorithm

b. Kruskal's algorithm



Suggested Reading

1. Data Structures and Algorithms, A. Aho, J. Hopcroft, and J. Ullman, Computer Science and Information Processing Addison-Wesley, Reading, Massachusetts, 1st edition, (Jan 11, 1983)

2. "Data Structures Using C" by A K Sharma

3. "Data Structures and Program Design in C" by Kruse Robert L

4. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss

5. "Data Structures and Algorithms" by Alfred V Aho and Jeffrey D Ullman

1. Introduction to the Lab Manual for Data Structure Programming Practice

Welcome to the Data Structure Programming Practice Laboratory! This manual is designed to be your comprehensive guide throughout the lab sessions, providing detailed instructions and essential information for each programming exercise. It is an indispensable tool that will aid you in mastering the principles and techniques required to successfully implement and manipulate various data structures.

2. Purpose of the Lab Manual

The primary purpose of this lab manual is to provide a structured approach to learning through hands-on programming exercises. By following the procedures outlined in this manual, you will:

- Gain practical experience with the concepts and techniques discussed in lectures.
- Develop critical thinking and problem-solving skills in the context of data structures.
- Enhance your ability to design, implement, and analyse data structures.
- Learn to debug and optimise code effectively.
- Foster collaborative skills by working in pairs or small groups on complex programming tasks.

3. Safety and Best Practices

- While programming does not involve physical risks, adhering to best practices is crucial for success:
- Write Clean Code: Follow coding standards and best practices to write readable, maintainable, and efficient code.
- Test Thoroughly: Regularly test your code to catch and fix errors early.
- Version Control: Use version control systems like Git to manage your code and collaborate with others.
- Backup Your Work: Regularly backup your work to avoid data loss.
- Documentation: Document your code and procedures to make your work understandable to others and future you.

4. Preparing for the Lab

To make the most of your lab sessions, it is essential to come prepared. This includes:

- Reading the exercise in the lab manual before hand.
- Understanding the theoretical background and the purpose of the exercise.
- Completing any pre-lab assignments or questions.
- Setting up the required development environment and tools.

5. During the Lab

While working on programming exercises, maintain a high level of attention to detail and precision. Write, test, and debug your code iteratively, and ensure that your workspace is organised.

6. After the Lab

Post-lab activities often include analysing code performance, writing reports, and discussing results. Use the guidelines provided in the lab manual to present your findings clearly and concisely.

By following this lab manual, you will gain valuable hands-on experience and deepen your understanding of the data structures and algorithms covered in your course. We hope you find your time in the laboratory both educational and enjoyable. Good luck, and happy coding!

7. Structure of the Lab Manual

Each lab exercise in this manual is organised into the following sections:

1. **Objective**: A brief overview of the goals and expected outcomes of the exercise.
2. **Theory**: An explanation of the data structures and algorithms relevant to the exercise.
3. **Experiment**: Step-by-step instructions to complete the exercise, including code snippets and examples.
4. **Test Cases**: Sample inputs and expected outputs to verify the correctness of your implementations.

Block 5 - Part A

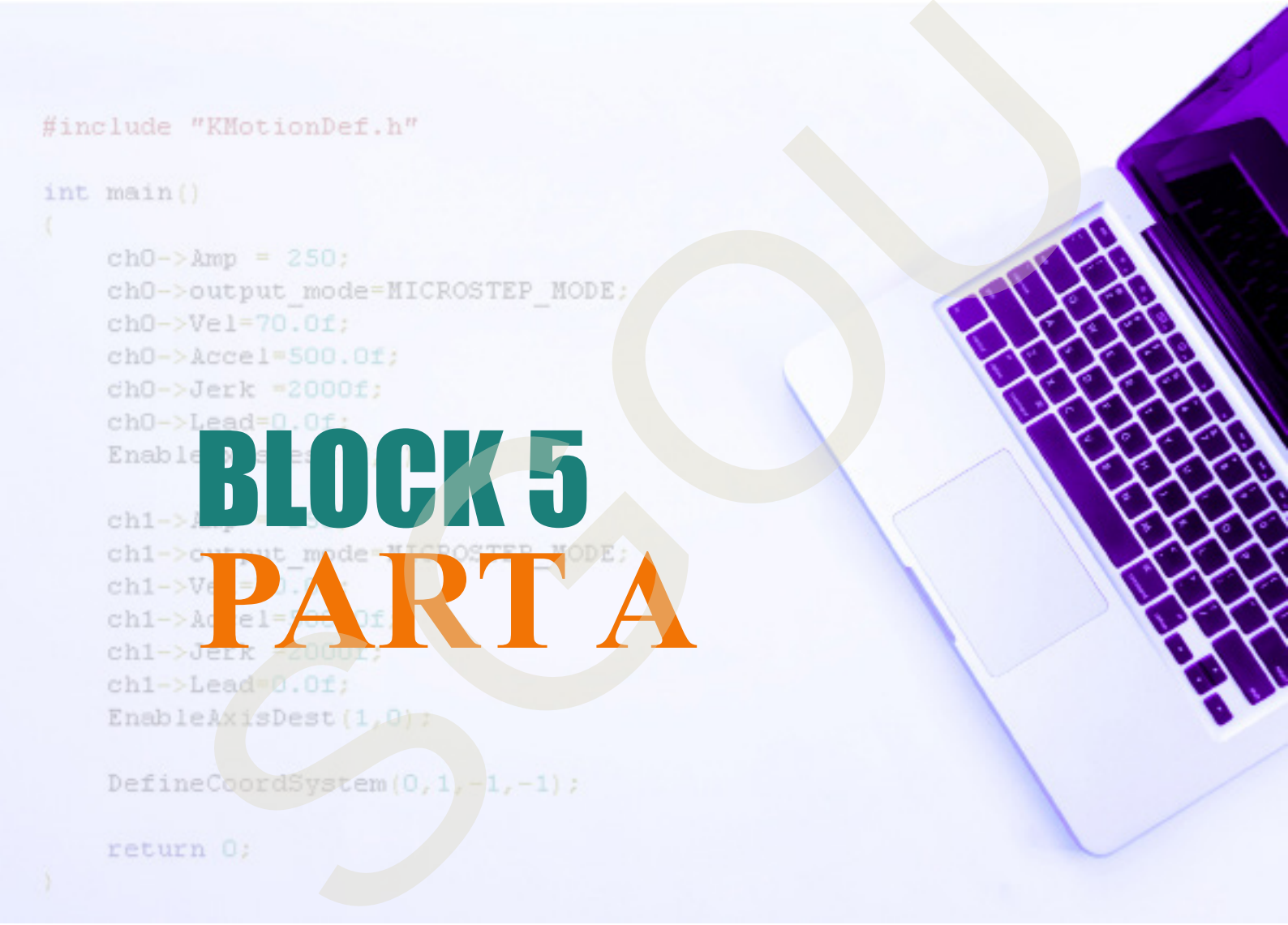
List of Experiments

1. Array initialization and display elements
2. Array operations
3. Linked list creation and display elements
4. Linked list operations
5. Queue operations using array
6. Queue operations using linked list
7. Stack operations using array
8. Stack operations using linked list

Block 6 - Part B

List of Experiments

1. Linear Search
2. Insertion Sort
3. Selection Sort
4. Binary Search
5. Quick sort



BLOCK 5
PART A

Array Initialization and Display Elements

5.1.1 Objective

Input statements enable programs to receive data from users or external sources, while output statements allow programs to communicate results, messages, or information back to the user. This lab session is designed to provide you with hands-on experience in working with input and output statements.

5.1.2 Theory

- To study the use of Control Strings in C (\a, \t, \n,..etc.)
- To familiarize the format specifiers in C
- To prompt and input basic details of a person (Name, age, sex)
- To print various message outputs using various format specifiers

5.1.3 Sub Experiment

Write a program to initialise and display elements in an array.

Algorithm:

STEP 1: Start

STEP 2: Initialise arr[] = {1, 2, 3, 4, 5}.

STEP 3: length= sizeof(arr)/sizeof(arr[0])

STEP 4: Print "Elements of given array:"

STEP 5: i=0. Repeat STEP 6 and STEP 7 UNTIL i<length

STEP 6: Print arr[i]

STEP 7: i = i+1

STEP 8: End

Test Cases

1. Initialise an array with a fixed size and predefined values.

Description: Test the basic functionality by initialising an array of a fixed size with specific integers.

Sample Input: Array of size 5 with elements {1, 2, 3, 4, 5}.

2. Array with Negative Numbers

Description: Test the array with a mixture of positive and negative integers.

Sample Input: Array of size 6 with elements {5, -3, 8, -1, 0, 7}.

3. Store the array with less number of elements than the declared upper bound.

Description: Initialize array with

int a[10]={1,2,3,8.9}; and display the array elements with indexes till a[11].

Expected output:

a[0]=1

a[1]=2

a[2]=3

a[3]=8

a[4]=0

a[5]=0

a[6]=0

a[7]=0

a[8]=0

a[9]=0

a[10]=279235328

a[11]=1293533327

4. Displaying Temperature Readings

Description: A meteorologist wants to store and display daily afternoon temperature readings for a week.

EXPERIMENT 5.2 Array operations

5.2.1 Objective

To implement various array operations such as searching, insertion and deletion.

5.2.2 Theory

Implementing various array operations involves creating functions to efficiently handle searching, insertion, and deletion of elements within the array. These operations enable dynamic data manipulation, ensuring optimal performance and adaptability for different applications.

5.2.3 Sub Experiments

1. Implement a program for searching elements in an array

Algorithm:

1. Start.
2. Input: An array arr of size n and the element x to be searched.
3. Set: found to false.
4. For i = 0 to n-1 do:

- If arr[i] equals x:
  - Print "Element found at index i".
  - Set found to true.
  - Break.

5. If found is false, print "Element not found".
6. Stop.

2. Implement a program for insertion of elements in an array

Algorithm:

1. Start
2. Input: An array arr of size n, the element x to be inserted, and the position pos where x is to be inserted.
3. Check: If pos is less than 0 or greater than n:
   - Print "Invalid position".
   - Stop.
4. For i = n-1 to pos -1 do:
   - arr[i+1] = arr[i]; // move the array element at the i^{th} position to $i+1^{th}$ position
5. Set arr[pos] to x.
6. Increment the size of the array n by 1.
7. Print the updated array.
8. Stop.

3. Implement a program for deletion of elements in an array

Algorithm:

1. Start.
2. Input: An array arr of size n and the position pos of the element to be deleted.
3. Check: If pos is less than 0 or greater than or equal to n:
   - Print "Invalid position".
   - Stop.
4. For i = pos to n-2 do:
   - Set arr[i] to arr[i+1].
5. Decrement the size of the array n by 1.
6. Print the updated array.
7. Stop.

Case Study

Test Cases

- **Array**: Verify that insertion of a particular element done correctly, searching for existing and non-existing elements returns the correct index.

Arrays				
Test Case	**Input Array**	**Operation**	**Target Element**	**Expected Result**
1	{4, 2, 1, 3}	Insertion at 3rd position	5	The value 5 Inserted at position 3
2	{4, 2, 1, 5, 3}	Deletion	3	Element 3 deleted
3	{}	Searching	5	-1
4	{1, 2, 3, 4, 5}	Searching	1	0
5	{10, 20, 30, 40, 50}	Searching	30	2

EXPERIMENT 5.3

Linked List Creation and Display Elements

5.3.1 Objective

To implement a singly linked list by creating nodes, inserting nodes and displaying its elements.

5.3.2 Theory

To implement a singly linked list, nodes are created dynamically to store data and a reference to the next node. Insertion operations add new nodes at the end of the list, and a traversal operation iterates through the list to display its elements.

5.3.3 Sub Experiments

1. Implement a program for creation, insertion and displaying elements in a linked list.

Algorithm:

1. Start.
2. Define a structure Node with two members:
   - int data: to store the integer data.
   - Node* next: to store the address of the next node.
3. Initialize head to NULL (indicating the start of the linked list).
4. Function to Create a New Node:
   - Input: int value.
   - Create a new node temp.
   - Set temp->data to value.
   - Set temp->next to NULL.
   - Return temp.

5. Function to Insert a Node at the End:
    - Input: int value.
    - Create a new node newNode using the function to create a new node.
    - If head is NULL, set head to newNode.
    - Else, traverse the list to find the last node.
    - Set the next of the last node to newNode.
6. Function to Display Elements:
    - Input: head.
    - Initialize current to head.
    - While current is not NULL:
        - Print current->data.
        - Move current to the next node.
7. Insert elements into the linked list by calling the insertion function.
8. Display the elements of the linked list by calling the display function.
9. Stop.

Test Cases

Linked Lists				
Test Case	**Linked List Elements**	**Operation**	**Target Element**	**Expected Result**
1	**4 -> 2 -> 1 -> 3**	**Insertion**	**5**	**The value 5 Inserted**
2	**4 -> 2 -> 1 -> 5 -> 3 -> 7**	**Insertion**	**7**	**The value 7 Inserted**

EXPERIMENT 5.4 Linked list operations

5.4.1 Objective

To implement Linked List operations such as insertion at middle, deletion and searching of elements.

5.4.2 Theory

To implement linked list operations such as insertion at the middle, deletion, and searching, one must efficiently manage node pointers to ensure the correct order and integrity of the list. These operations involve traversing the list to locate the target positions, modifying node connections for insertion or deletion, and iterating through the nodes for searching specific elements.

5.4.3 Sub Experiments

1. Implement a program for Node creation, insertion at middle, deletion of elements and searching an element in a linked list.

Algorithm:

1. **Structure Definition**

   1. Start.
   2. Define a structure Node with two members:
      - int data: to store the integer data.
      - Node* next: to store the address of the next node.

2. **Insertion at middle**

   3. Function to Insert at middle:
      - Input: int value, int position.
      - Create a new node newNode and set its data to value.

- If head is NULL or position is 0, set newNode->next to head and head to newNode.
- Else, traverse the list to find the node just before the desired position.
- Set newNode->next to the next of the found node.
- Set the next of the found node to newNode.

3. Deletion

4. Function to Delete a Node:

- Input: int value.
- If head is NULL, print "List is empty".
- If head->data is value, set head to head->next and free the old head.
- Else, traverse the list to find the node with value.
- If found, set the next of the previous node to the next of the current node and free the current node.
- If not found, print "Value not found".

4. Searching

5. Function to Search a Node:

- Input: int value.
- Initialise current to head and position to 0.
- While current is not NULL:
  - If current->data is value, print "Value found at position" and position, then return.
  - Move current to the next node and increment position.
- If current becomes NULL, print "Value not found".

6. Insert elements into the linked list by calling the insertion function.

7. Delete elements from the linked list by calling the deletion function.

8. Search for elements in the linked list by calling the search function.

9. Stop.

Test Cases

Linked List: Ensure that the linear search correctly identifies the index of the target element

Linked Lists				
Test Case	**Linked List Elements**	**Operation**	**Target Element**	**Expected Result**
1	4 -> 2 -> 1 -> 3	Insertion at 3rd position	5	The value 5 Inserted
2	4 -> 2 -> 1 -> 5 -> 3 -> 7	Insertion at 5th position	7	The value 7 Inserted
3	(empty list)	Deletion	5	Element not found
4	1 -> 2 -> 3 -> 4 -> 5	Deletion	1	Element 1 deleted from list
4	1->2->3->4->5	Searching	1	0
5	10->20->30->40->50	Searching	30	2

EXPERIMENT 5.5

Queue Operations Using Array

5.5.1 Objective

To implement various queue operation such as insertion, deletion and searching using array

5.5.2 Theory

Initialization:

- An array queue of size MAX is used to store the elements.
- Two pointers front and rear are initialised to -1 to represent an empty queue.

Insertion (Enqueue):

- Check if the queue is full by comparing rear with MAX - 1.
- If the queue is empty (front == -1), initialise front to 0.
- Increment rear and insert the value at queue[rear].

Deletion (Dequeue):

- Check if the queue is empty by comparing front and rear.
- Store the value at queue[front] in a temporary variable and increment front.
- If the queue becomes empty after the deletion, reset front and rear to -1.

Searching:

- Check if the queue is empty.
- Traverse the queue from front to rear and compare each element with the search value.
- Print the position if the value is found; otherwise, print that the value is not found.

5.5.3 Sub Experiments

1. Implement a program for initialisation, insertion (enqueue), deletion of elements (dequeue) and searching an element in a queue using an array.

Algorithm:

1. **Initialise the Queue**
   - Start.
   - Define an array queue of size MAX.
   - Define two variables front and rear and initialize them to -1.
2. Insertion (Enqueue) Operation

   1. Function enqueue(int value):
   - Step 1: Check if the queue is full.
     - If (rear == MAX - 1), print "Queue is full" and return.
   - Step 2: If the queue is empty, set front = 0.
   - Step 3: Increment rear by 1.
   - Step 4: Set queue[rear] = value.
3. **Deletion (Dequeue) Operation**

   2. Function dequeue():
   - Step 1: Check if the queue is empty.
     - If (front == -1 || front > rear), print "Queue is empty" and return -1.
   - Step 2: Store the value queue[front] in a variable temp.
   - Step 3: Increment front by 1.
   - Step 4: If front becomes greater than rear, reset front and rear to -1.
   - Step 5: Return temp.
4. **Searching Operation**

   3. Function search(int value):
   - Step 1: Check if the queue is empty.
     - If (front == -1 || front > rear), print "Queue is empty" and return -1.
   - Step 2: Traverse the queue from front to rear.

- Step 3: If queue[i] = = value, print "Value found at position i" and return i.
- Step 4: If value is not found, print "Value not found" and return -1.

5. Stop.

Test Cases

Verify that insertion of a particular element done correctly, searching for existing and non-existing elements returns the correct index.

Queue operations using Array

Test Case	Input Array	Operation	Target Element	Expected Result
1	{4, 2, 1, 3}	(Enqueue) Insertion	5	The value 5 Inserted
2	{4, 2, 1, 5, 3}	(Dequeue) Deletion	3	Element 3 deleted
3	{}	Searching	5	-1
4	{1, 2, 3, 4, 5}	Searching	1	0
5	{10, 20, 30, 40, 50}	Searching	30	2

EXPERIMENT 5.6

Queue operations using linked list

5.6.1 Objective

To implement various queue operations such as insertion, deletion and searching using linked lists.

5.6.2 Theory

Initialization:

- A structure Node is defined with an integer data and a pointer to the next node.
- Two pointers front and rear are initialized to NULL to represent an empty queue.

Insertion (Enqueue):

- A new node is created and its data is set to the value to be inserted.
- If the queue is empty (rear == NULL), both front and rear are set to the new node.
- Otherwise, the new node is added at the end of the queue and rear is updated.

Deletion (Dequeue):

- The function checks if the queue is empty and returns -1 if true.
- The value of the front node is stored and front is moved to the next node.
- If the queue becomes empty after deletion, rear is set to NULL.
- The old front node is freed and the stored value is returned.

Searching:

- The function checks if the queue is empty and returns -1 if true.
- The queue is traversed from front to rear, and the position of the value is printed if found.
- If the value is not found, a message is printed and -1 is returned.

5.6.3 Sub Experiments

1. Implement a program for initialization of elements, insertion (enqueue), deletion of elements (dequeue) and searching an element in a queue using a linked list.

Algorithm:

1. Initialize the Queue
   - ♦ Start.
   - ♦ Define a structure for the Node with data and a pointer to the next Node.
   - ♦ Define two pointers front and rear and initialize them to NULL.
2. Insertion (Enqueue) Operation

   1. Function enqueue(int value):
   - ♦ Step 1: Create a new node and assign the value to the node.
   - ♦ Step 2: If rear is NULL, set both front and rear to the new node.
   - ♦ Step 3: Else, set rear->next to the new node and update rear to the new node.
3. Deletion (Dequeue) Operation

   2. Function dequeue():
   - ♦ Step 1: Check if the queue is empty (front is NULL).
     - ♦ If true, print "Queue is empty" and return -1.
   - ♦ Step 2: Store the value of front->data in a variable tempValue.
   - ♦ Step 3: Move front to the next node.
   - ♦ Step 4: If front becomes NULL, set rear to NULL.
   - ♦ Step 5: Free the old front node.
   - ♦ Step 6: Return tempValue.
4. Searching Operation

   3. Function search(int value):
   - ♦ Step 1: Check if the queue is empty (front is NULL).
     - ♦ If true, print "Queue is empty" and return -1.
   - ♦ Step 2: Initialize a temporary pointer temp to front and a position counter pos to 0.

- Step 3: Traverse the queue until temp is NULL.
    - If temp->data equals value, print "Value found at position pos" and return pos.
    - Move temp to the next node and increment pos.
- Step 4: If value is not found, print "Value not found" and return -1.

4. Stop.

Test Cases

Queue operations using Linked Lists

Test Case	Linked List Elements	Operation	Target	Expected Result
1	4 -> 2 -> 1 -> 3	(Enqueue) Insertion	5	The value 5 Inserted
2	4 -> 2 -> 1 -> 5 -> 3 -> 7	(Enqueue) Insertion	7	The value 7 Inserted
3	(empty list)	(Dequeue) Deletion	5	Element not found
4	1 -> 2 -> 3 -> 4 -> 5	(Dequeue) Deletion	1	Element 1 deleted from list
4	1->2->3->4->5	Searching	1	0
5	10->20->30->40->50	Searching	30	2

Stack Operations using Array

5.7.1 Objective

- To implement various stack operations push, pop, isEmpty, isFull using arrays.

5.7.2 Theory

Initialization:

- The stack is represented as an array stack with a fixed size SIZE.
- The variable top keeps track of the top element in the stack, initialised to -1 to indicate an empty stack.

Push Operation:

- Checks if the stack is full (top == SIZE - 1). If it is, it prints "Stack Overflow".
- Otherwise, it increments top and assigns the value to stack[top].

Pop Operation:

- Checks if the stack is empty (top == -1). If it is, it prints "Stack Underflow" and returns -1.
- Otherwise, it returns the value of stack[top] and decrements top.

IsEmpty Operation:

- Checks if the stack is empty (top == -1). Returns 1 if true, otherwise 0.

IsFull Operation:

- Checks if the stack is full (top == SIZE - 1). Returns 1 if true, otherwise 0.

5.7.3 Sub Experiments

1. Implement a program for initialization of elements, insertion (push), deletion of elements (pop) , check IsEmpty and check IsFull operations in a stack using an array.

Algorithm:

1. Initialize the Stack
   - Start.
   - Define an array stack of a fixed size and an integer variable top initialized to -1 to represent the top of the stack.
2. Push Operation

   1. Function push(int value):
   - Step 1: Check if the stack is full (top == size - 1).
     - If true, print "Stack Overflow" and return.
   - Step 2: Increment top by 1.
   - Step 3: Assign the value to stack[top].
3. Pop Operation

   2. Function pop():
   - Step 1: Check if the stack is empty (top == -1).
     - If true, print "Stack Underflow" and return -1.
   - Step 2: Store the value of stack[top] in a variable temp.
   - Step 3: Decrement top by 1.
   - Step 4: Return temp.
4. IsEmpty Operation

   3. Function isEmpty():
   - Step 1: Check if the stack is empty (top == -1).
     - If true, return 1.
     - Else, return 0.

5. IsFull Operation

   4. Function isFull():

   - Step 1: Check if the stack is full (top == size - 1).
     - If true, return 1.
     - Else, return 0.

6. Display Operation

   5. Function display():

   - Step 1: Check if the stack is empty (top == -1).
     - If true, print "Stack is empty" and return.
   - Step 2: Initialize a loop variable i to top.
   - Step 3: Loop from i to 0, printing stack[i].

7. Stop

Test Cases

Verify that Push operation of a particular element done correctly, searching for existing and non-existing elements returns the correct index.

Stack operations using Array

Test Case	Input Array	Operation	Target Element	Expected Result
1	{4, 2, 1, 3}	(Push) Insertion	5	The value 5 pushed into the stack
2	{4, 2, 1, 5, 3}	(POP) Deletion	3	Element 3 popped
3	{}	IsEmpty()		-1
4	{1, 2, 3, 4, 5}	IsFull()		Not possible to Push element into the stack
5	{10, 20, 30, 40, 50}	Searching	30	2

EXPERIMENT 5.8 Stack operations using linked list

5.8.1 Objective

To implement various stack operations push, pop, isEmpty using linked lists.

5.8.2 Theory

Initialization:

- The stack is represented using a linked list. Each node has an int data and a Node* next.
- The top pointer points to the top of the stack, initially set to NULL.

Push Operation:

- A new node is created and its data is assigned the given value.
- The new node's next pointer is set to the current top.
- The top pointer is updated to point to the new node.

Pop Operation:

- Checks if the stack is empty (top == NULL). If it is, it prints "Stack Underflow" and returns -1.
- Otherwise, it stores the value of top->data, updates the top pointer to top->next, frees the memory of the old top node, and returns the stored value.

IsEmpty Operation:

- Checks if the stack is empty (top == NULL). Returns 1 if true, otherwise 0.

5.8.3 Sub Experiments

1. Implement a program for initialization of elements, insertion (push), deletion of elements (pop) and check IsEmpty operations in a stack using a linked list.

Algorithm:

1. Initialize the Stack
   - ♦ Start.
   - ♦ Define a structure Node with two members: int data and Node* next.
   - ♦ Define a pointer top initialized to NULL to represent the top of the stack.
2. Push Operation

   1. Function push(int value):
      - ♦ Step 1: Create a new node newNode and allocate memory for it.
      - ♦ Step 2: Assign value to newNode->data.
      - ♦ Step 3: Set newNode->next to top.
      - ♦ Step 4: Set top to newNode.
3. Pop Operation

   2. Function pop():
      - ♦ Step 1: Check if the stack is empty (top == NULL).
        - ♦ If true, print "Stack Underflow" and return -1.
      - ♦ Step 2: Store the value of top->data in a variable temp.
      - ♦ Step 3: Create a pointer tempNode and set it to top.
      - ♦ Step 4: Set top to top->next.
      - ♦ Step 5: Free the memory of tempNode.
      - ♦ Step 6: Return temp.
4. IsEmpty Operation

   3. Function isEmpty():
      - ♦ Step 1: Check if the stack is empty (top == NULL).
        - ♦ If true, return 1.
        - ♦ Else, return 0.

5. Display Operation

4. Function display():

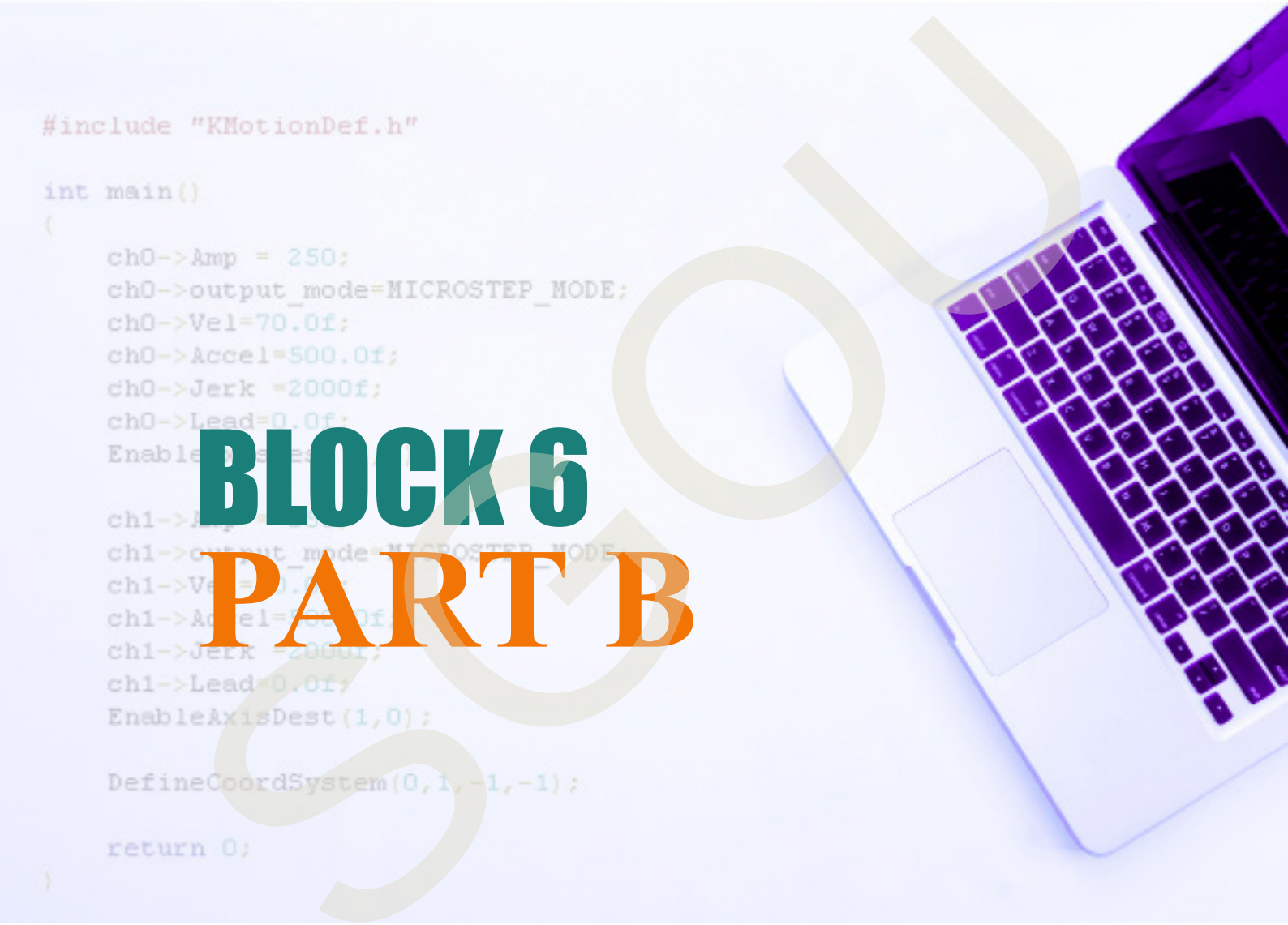
- Step 1: Check if the stack is empty (top == NULL).
  - If true, print "Stack is empty" and return.
- Step 2: Create a pointer current and set it to top.
- Step 3: While current is not NULL:
  - Print current->data.
  - Set current to current->next.

6. Stop

Test Cases

Stack operations using Linked Lists

Test Case	Linked List Elements	Operation	Target	Expected Result
1	4 -> 2 -> 1 -> 3	(Push) Insertion	5	The value 5 Inserted
2	4 -> 2 -> 1 -> 5 -> 3 -> 7	(Push) Insertion	7	The value 7 Inserted
3	(empty list)	IsEmpty()		-1
4	1 -> 2 -> 3 -> 4 -> 5	(POP) Deletion	1	Element 1 deleted from list
4	1->2->3->4->5	Searching	1	0
5	10->20->30->40->50	Searching	30	2



BLOCK 6
PART B

EXPERIMENT 6.1 Linear Search

6.1.1 Objective

- Implement linear search on an array of integers.
- Display the number of comparisons required to search an element.
- Implement linear search on a linked list, queue, and stack.

6.1.2 Theory

Linear search is a simple search algorithm that checks each element in the array until the target element is found or the end of the array is reached. It has a time complexity of O(n).

6.1.3 Sub Experiments

1. Linear Search on Array:

Write a program to implement linear search on an array of integers.

Display the number of comparisons required to search for an element.

Algorithm:

Step 1: Initialize the array with integer elements.

Step 2: Prompt the user to input the target element.

Step 3: Iterate through the array elements one by one.

Step 4: Compare each element with the target element.

Step 5: Count and display the number of comparisons made.

Step 6: If the target element is found, return its index; otherwise, indicate it is not in the array.

2. Linear Search on Linked List:

- Implement linear search on a linked list and display the required number of comparisons.

Algorithm:

Step 1: Create a linked list with integer elements.

Step 2: Prompt the user to input the target element.

Step 3: Traverse the linked list node by node.

Step 4: Compare each node's data with the target element.

Step 5: Count and display the number of comparisons made.

Step 6: If the target element is found, return its position; otherwise, indicate that the element is not in the linked list.

3. **Linear Search on Queue and Stack:**

- Implement linear search on a queue and stack using the respective data structures and display the number of comparisons required.

Algorithm:

Step 1: Initialize a queue and a stack with integer elements.

Step 2: Prompt the user to input the target element.

Step 3: Traverse the queue and stack element by element.

Step 4: Compare each element with the target element.

Step 5: Count and display the number of comparisons made.

Step 6: If the target element is found, return its position; otherwise, indicate that the element is not in the queue or stack.

Test Cases

- **Array**: Verify that searching for existing and non-existing elements returns the correct index.
- **Linked List**: Ensure that the linear search correctly identifies the index of the target element.
- **Queue and Stack**: Confirm that the search works correctly and returns the expected index.

In Arrays

Test Case	Input Array	Target	Expected Result
1	{4, 2, 1, 5, 3}	5	3
2	{4, 2, 1, 5, 3}	6	-1
3	{}	5	-1
4	{1, 2, 3, 4, 5}	1	0
5	{10, 20, 30, 40, 50}	30	2

In Linked Lists

Test Case	Linked List Elements	Target	Expected Result
1	4 -> 2 -> 1 -> 5 -> 3	5	3
2	4 -> 2 -> 1 -> 5 -> 3	6	-1
3	(empty list)	5	-1
4	1 -> 2 -> 3 -> 4 -> 5	1	0
5	10 -> 20 -> 30 -> 40 -> 50	30	2

In Queues

Test Case	Queue Elements	Target	Expected Result
1	{4, 2, 1, 5, 3}	5	3
2	{4, 2, 1, 5, 3}	6	-1
3	{}	5	-1
4	{1, 2, 3, 4, 5}	1	0
5	{10, 20, 30, 40, 50}	30	2

In Stacks

Test Case	Stack Elements	Target	Expected Result
1	{4, 2, 1, 5, 3}	5	3
2	{4, 2, 1, 5, 3}	6	-1
3	{}	5	-1
4	{1, 2, 3, 4, 5}	1	0
5	{10, 20, 30, 40, 50}	30	2

Insertion Sort

6.2.1 Objective

- Implement insertion sort on an array of integers.
- Display the sorted array.
- Implement insertion sort on a linked list, queue, and stack.

6.2.2 Theory

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It has a time complexity of O(n^2) in the worst case but performs well on small or nearly sorted arrays.

6.2.3 Sub Experiments

1. Insertion Sort on Array:

- Write a program to implement insertion sort on an array of integers. Also display the sorted array.

Algorithm:

Step 1: Initialize the array with integer elements.

Step 2: Iterate from the second element to the last element.

Step 3: For each element, compare it with the elements before it.

Step 4: Shift the larger elements to the right.

Step 5: Insert the current element into its correct position.

Step 6: Display the sorted array.

Insertion Sort on Linked List:

- Implement insertion sort on a linked list and display the sorted list.

Algorithm:

Step 1: Create a linked list with integer elements.

Step 2: Traverse the linked list from the second node to the last node.

Step 3: For each node, compare its data with the nodes before it.

Step 4: Shift the larger nodes to the right.

Step 5: Insert the current node into its correct position.

Step 6: Display the sorted linked list.

3. **Insertion Sort on Queue and Stack:**

- Implement insertion sort on a queue and stack using the respective data structures and display the sorted data.

Algorithm:

Step 1: Initialize a queue and a stack with integer elements.

Step 2: Traverse the queue and stack from the second element to the last ele ment.

Step 3: For each element, compare it with the elements before it.

Step 4: Shift the larger elements to the right.

Step 5: Insert the current element into its correct position.

Step 6: Display the sorted queue and stack.

Test Cases

- **Array**: Verify that the array is sorted correctly.
- **Linked List**: Ensure that the linked list is sorted correctly.
- **Queue and Stack**: Confirm that the data is sorted correctly in the respective data structures.

1. Insertion Sort in Arrays

Test Cases

Test Case	Input Array	Expected Result
1	{4, 2, 1, 5, 3}	{1, 2, 3, 4, 5}
2	{5, 4, 3, 2, 1}	{1, 2, 3, 4, 5}
3	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
4	{1}	{1}
5	{9, 7, 5, 3, 1}	{1, 3, 5, 7, 9}

2. Insertion Sort in Linked Lists

Test Case	Input Linked List	Expected Result
1	4 -> 2 -> 1 -> 5 -> 3	1 -> 2 -> 3 -> 4 -> 5
2	5 -> 4 -> 3 -> 2 -> 1	1 -> 2 -> 3 -> 4 -> 5
3	1 -> 2 -> 3 -> 4 -> 5	1 -> 2 -> 3 -> 4 -> 5
4	1	1
5	9 -> 7 -> 5 -> 3 -> 1	1 -> 3 -> 5 -> 7 -> 9

3. Insertion Sort in Queues

Test Case	Input Queue	Expected Result
1	{4, 2, 1, 5, 3}	{1, 2, 3, 4, 5}
2	{5, 4, 3, 2, 1}	{1, 2, 3, 4, 5}
3	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
4	{1}	{1}
5	{9, 7, 5, 3, 1}	{1, 3, 5, 7, 9}

4. Insertion Sort in Stacks

Test Cases

Test Case	Input Stack	Expected Result
1	{4, 2, 1, 5, 3}	{1, 2, 3, 4, 5}
2	{5, 4, 3, 2, 1}	{1, 2, 3, 4, 5}
3	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
4	{1}	{1}
5	{9, 7, 5, 3, 1}	{1, 3, 5, 7, 9}

EXPERIMENT 6.3 Selection Sort

6.3.1 Objective

- Implement selection sort on an array of integers.
- Display the sorted array.
- Implement selection sort on a linked list, queue, and stack.

6.3.2 Theory

Selection sort is a simple sorting algorithm that repeatedly selects the minimum element from the unsorted portion of the array and swaps it with the first unsorted element. It has a time complexity of O(n^2).

6.3.3 Sub Experiments

1. Selection Sort on Array:

- Write a program to implement selection sort on an array of integers.
- Display the sorted array.

Algorithm:

Step 1: Initialize the array with integer elements.

Step 2: Iterate through the array elements.

Step 3: For each element, find the minimum element in the unsorted portion.

Step 4: Swap the minimum element with the first unsorted element.

Step 5: Display the sorted array.

2. Selection Sort on Linked List:

Implement selection sort on a linked list and display the sorted list.

Algorithm:

Step 1: Create a linked list with integer elements.

Step 2: Traverse the linked list to find the minimum node in the unsorted portion.

Step 3: Swap the data of the minimum node with the first unsorted node.

Step 4: Repeat until the linked list is sorted.

Step 5: Display the sorted linked list.

3. Selection Sort on Queue and Stack:

- Implement selection sort on a queue and stack using the respective data structures and display the sorted data.

Algorithm:

Step 1: Initialize a queue and a stack with integer elements.

Step 2: Traverse the queue and stack to find the minimum element in the unsorted portion.

Step 3: Swap the minimum element with the first unsorted element.

Step 4: Repeat until the queue and stack are sorted.

Step 5: Display the sorted queue and stack.

Test Cases

- Array: Verify that the array is sorted correctly.
- Linked List: Ensure that the linked list is sorted correctly.
- Queue and Stack: Confirm that the data is sorted correctly in the respective data structures.

Test Cases for Selection Sort		
Arrays:		
Test Case	**Input Array**	**Expected Output**
1	{64, 25, 12, 22, 11}	{11, 12, 22, 25, 64}
2	{5, 4, 3, 2, 1}	{1, 2, 3, 4, 5}
3	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
4	{99, 88, 77, 66, 55, 44}	{44, 55, 66, 77, 88, 99}
5	{1}	{1}
Linked Lists:		
Test Case	**Input Linked List**	**Expected Output**
1	4 -> 2 -> 1 -> 5 -> 3	1 -> 2 -> 3 -> 4 -> 5
2	5 -> 4 -> 3 -> 2 -> 1	1 -> 2 -> 3 -> 4 -> 5
3	1 -> 2 -> 3 -> 4 -> 5	1 -> 2 -> 3 -> 4 -> 5
4	1	1
5	9 -> 7 -> 5 -> 3 -> 1	1 -> 3 -> 5 -> 7 -> 9
Queues:		
Test Case	**Input Queue**	**Expected Output**
1	{64, 25, 12, 22, 11}	{11, 12, 22, 25, 64}
2	{5, 4, 3, 2, 1}	{1, 2, 3, 4, 5}
3	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
4	{99, 88, 77, 66, 55, 44}	{44, 55, 66, 77, 88, 99}
5	{1}	{1}
Stacks:		
Test Case	**Input Stack**	**Expected Output**
1	{64, 25, 12, 22, 11}	{11, 12, 22, 25, 64}
2	{5, 4, 3, 2, 1}	{1, 2, 3, 4, 5}
3	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
4	{99, 88, 77, 66, 55, 44}	{44, 55, 66, 77, 88, 99}
5	{1}	{1}

EXPERIMENT 6.4 Binary Search

6.4.1 Objective

- Implement binary search on an array of integers.
- Display the number of comparisons required to search an element.
- Implement binary search on a binary search tree.

6.4.2 Theory

Binary search is an efficient search algorithm that works on sorted arrays. It repeatedly divides the search interval in half and has a time complexity of O(log n).

6.4.3 Sub Experiments

1. Binary Search on arrays, linked lists, queues and stacks

- Write a program to implement binary search on an array of integers.
- Implement binary search on linked lists, queues (implemented as arrays), and stacks (implemented as arrays).
- Display the number of comparisons required to search for an element.

Algorithm:

Step 1: Initialize an array with integer elements.

Step 2: Prompt the user to input the target element.

Step 3: Set the low and high pointers to the beginning and end of the array.

Step 4: While the low pointer is less than or equal to the high pointer, calculate the mid pointer.

Step 5: Compare the element at the mid pointer with the target element.

Step 6: If the target element is found, return its index; otherwise, adjust the low or high pointer.

Step 7: Count and display the number of comparisons made.

2. Binary Search on Binary Search Tree:

- Implement binary search on a binary search tree and display the number of comparisons required.

Algorithm:

Step 1: Create a binary search tree with integer elements.

Step 2: Prompt the user to input the target element.

Step 3: Start from the root node and compare the target element with the node's data.

Step 4: If the target element is equal to the node's data, return the node.

Step 5: If the target element is less than the node's data, move to the left child; otherwise, move to the right child.

Step 6: Repeat steps 3 to 5 until the target element is found or a leaf node is reached.

Step 7: Count and display the number of comparisons made.

Test Cases

Array:Verify its correctness using various test cases (sorted arrays with different sizes, target values inside and outside the array).

Test Cases for Binary Search			
Arrays:			
Test Case	**Input Array**	**Element to Search**	**Expected Output**
1	{1, 2, 3, 4, 5}	3	Found at index 2
2	{11, 22, 33, 44, 55}	44	Found at index 3
3	{5, 10, 15, 20, 25}	100	Not Found
4	{1, 3, 5, 7, 9}	4	Not Found
5	{2}	2	Found at index 0
Linked Lists:			
Test Case	**Input Linked List**	**Element to Search**	**Expected Output**
1	1 -> 2 -> 3 -> 4 -> 5	3	Found at index 2
2	5 -> 10 -> 15 -> 20 -> 25	100	Not Found
3	11 -> 22 -> 33 -> 44 -> 55	44	Found at index 3
4	1	1	Found at index 0
5	7 -> 14 -> 21 -> 28 -> 35	14	Found at index 1
Queues:			
Test Case	**Input Queue**	**Element to Search**	**Expected Output**
1	{1, 2, 3, 4, 5}	3	Found at index 2
2	{5, 10, 15, 20, 25}	100	Not Found
3	{11, 22, 33, 44, 55}	44	Found at index 3
4	{1, 3, 5, 7, 9}	4	Not Found
5	{2}	2	Found at index 0
Stacks:			
Test Case	**Input Stack**	**Element to Search**	**Expected Output**
1	{1, 2, 3, 4, 5}	3	Found at index 2
2	{5, 10, 15, 20, 25}	100	Not Found
3	{11, 22, 33, 44, 55}	44	Found at index 3
4	{1, 3, 5, 7, 9}	4	Not Found
5	{2}	2	Found at index 0

Binary Search Tree: Ensure the binary search correctly identifies the target element and the number of comparisons.

Basic Cases:

BST:
markdown

Copy code

```
      8
     / \
    3   10
   / \   \
  1   6   14
     / \  /
    4   7 13
```

Test Case 1:

- Search Target: 6
- Expected Output: Node with value 6 found.

Test Case 2:

- Search Target: 14
- Expected Output: Node with value 14 found.

Test Case 3:

- Search Target: 3
- Expected Output: Node with value 3 found.

Test Case 4:

- Search Target: 1
- Expected Output: Node with value 1 found.

Test Case 5:

- Search Target: 15
- Expected Output: Node with value 15 not found.

Edge Cases:

Test Case 6:

- BST: (empty BST)
- Search Target: 5
- Expected Output: Node with value 5 not found.

Note:

- Ensure that the algorithm handles edge cases such as an empty BST, single-node BST, and nodes not present in the BST.
- Consider visualizing the BST structure or printing the traversal steps to aid in understanding and debugging.

EXPERIMENT 6.5 Quicksort

6.5.1 Objective

- Implement quicksort on an array of integers.
- Display the sorted array.

6.5.2 Theory

Quicksort is a highly efficient sorting algorithm that uses a divide-and-conquer strategy. It has an average time complexity of O(n log n).

6.5.3 Sub Experiments

1. Quicksort on Array:

- Write a program to implement quicksort on an array of integers in ascending /descending order and display the sorted array.

Algorithm:

Step 1: Initialize the array with integer elements.

Step 2: Choose a pivot element from the array.

Step 3: Partition the array into three parts: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot.

Partitioning Procedure:

- Choose a pivot element from the array. Common choices include the first element, the last element, or a randomly chosen element.
- Rearrange the elements such that all elements less than the pivot are moved to its left, and all elements greater than the pivot are moved to its right.
- Return the index of the pivot after partitioning.

Step 4: Recursively apply the above steps to the sub-arrays.

Step 5: Combine the sorted subarrays and the pivot to get the final sorted array

2. Recursive Sorting:

- Recursively apply the quicksort function to the left and right subarrays of the pivot until the entire array is sorted.

Test Cases

Test Cases for Quicksort

1. **Basic Cases:**

- **Input:** [5, 3, 8, 4, 2, 7, 1]
  - **Expected Output:** [1, 2, 3, 4, 5, 7, 8]
- **Input:** [10, 20, 30, 40, 50]
  - **Expected Output:** [10, 20, 30, 40, 50]
- **Input:** [100, 20, 40, 30, 10]
  - **Expected Output:** [10, 20, 30, 40, 100]

2. **Edge Cases:**

- **Input:** [1]
  - **Expected Output:** [1]
- **Input:** [] (empty array)
  - **Expected Output:** []

3. **Performance Test:**

- **Input:** A large array with random elements (e.g., [87, 34, 92, 12, 56, 78, 45, ...])
  - **Expected Output:** Array sorted in non-decreasing order.

4. **Sorted Array:**

- **Input:** [1, 2, 3, 4, 5]
  - **Expected Output:** [1, 2, 3, 4, 5]

5. **Reverse Sorted Array:**

- **Input:** [5, 4, 3, 2, 1]
  - **Expected Output:** [1, 2, 3, 4, 5]

6. **Array with Duplicate Elements:**

- **Input:** [5, 2, 8, 1, 7, 2, 4, 3, 1]
  - **Expected Output:** [1, 1, 2, 2, 3, 4, 5, 7, 8]

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ……

Reg. No : .....

Name : …….....

Model Question Paper Set- I

End Semester Examination 2024

BACHELOR OF COMPUTER APPLICATION

BCA2024

B21CA04DC: DATA STRUCTURES

(CBCS - UG)

2023-24 - Admission Onwards

Time: 3 Hours Max Marks: 70

Section A

Answer any ten questions. Each carries one mark (10x1=10 marks)

1. What is the term used to define the step by step procedure to solve a problem?
2. Give an example for static data structure.
3. How do the elements of the array are referred to ?
4. What is the process of inserting an element into the stack?
5. Which part is used to get the first element from the circular queue?
6. Name the structure which contains a pointer to itself ?
7. Which pointer needs to be updated when inserting a node at the beginning of a singly linked list?
8. Name the data structures that maintain two pointers to store next and previous nodes.
9. What are the two pointers in the linked representation of a queue q consisting of a list and two pointers?
10. Which is the other name for leaf node?
11. Which traversal technique lists the nodes of a binary search tree in ascending order?

12. Which is the at most value of the difference between the heights of the left and right subtrees of all the nodes in a BST that makes the efficiency of searching is considered to be ideal?

13. What is the name of a graph which has either a self loop or parallel edges or both?

14. The property that the algorithm terminates after a finite number of steps is known as _____ .

15. What is the best case time complexity of insertion sort ?

Section B

Answer any five questions. Each carries two marks (5x2=10 marks)

16. What is stack in data structure? Name two stack operations.

17. What are enqueue() and dequeue() operations?

18. How do you access the data in a node?

19. What is a START pointer in a singly linked list?What is its purpose?

20. Differentiate cyclic and acyclic graphs .

21. Draw a complete binary tree with exactly six nodes.

22. What is the time complexity of an algorithm?

23. What are the features of an algorithm?

24. What are the applications of the minimax algorithm?

25. What do you mean by a spanning tree?

Section C

Answer any five questions. Each carries four marks (4x5=20 marks)

26. Evaluate the postfix expression AB+CD*/. Where A=1, b=1, C=3, D=2 .

27. Define a priority queue. How does it differ from a simple queue?

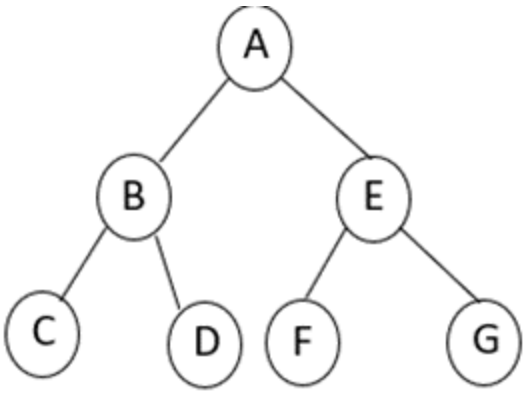
28. Describe the process of traversing a linked list to count the number of nodes.

29. Discuss the advantages of linked lists over arrays.

30. Write the algorithm for the push and pop operation in a stack implemented with a linked list.

31. Define a full binary tree and give an example.

32. Consider the following binary tree:



Write the order of the nodes visited in:

1) In-order traversal
2) Pre-order traversal
3) Post-order traversal

33. Differentiate between AVL trees and BST.

34. Explain the selection sort with examples.

35. Explain linear search algorithms with example and find time and space complexity.

Section D

Answer any two questions. Each carries fifteen marks (2x15=30 marks)

36. Describe how infix expressions are converted to postfix expressions? Convert the following infix expression into prefix expression

A - B - D * E / F + B * C

37. Describe the process of deleting nodes from

a. The beginning

b. From the end

c. From a specific position in a singly linked list.

38. Describe an algorithm and an illustration for the following:

i. The 'insertion of a node' into a BST.
ii. The 'deletion of a node' from a BST.

39. Design and Explain Prim's Algorithm for Finding the Minimum Spanning Tree (MST) of a Graph.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ……

Reg. No : .....

Name : …….....

Model Question Paper Set- II

End Semester Examination 2024

BACHELOR OF COMPUTER APPLICATION

BCA2024

B21CA04DC: DATA STRUCTURES

(CBCS - UG)

2023-24 - Admission Onwards

Time: 3 Hours Max Marks: 70

Section A

Answer any ten questions. Each carries one mark (10x1=10 marks)

1. What is the term used to refer to processed data called ?
2. Write an example for dynamic data structure.
3. The declaration int a[2][5] will allocate _____ bytes.
4. What is the process of removing elements from the stack?
5. identifies the data structure, which allows deletions as per the priority?
6. A linked list contains a data part and _____ part.
7. What do you need to update when deleting the last node in a singly linked list?
8. In a circular linked list, which node does the last node point to?
9. What operation is used for inserting an element “x” to the stack “s” ?
10. What is the maximum number of children that a binary tree can have?
11. Name one of the most efficient tree data structures?
12. What is the name of a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children?

13. What is the name of the graph which does not have self loop or parallel edges ?

14. _____is the property that each instruction of an algorithm is clear and unambiguous.

15. What is the space complexity of selection sort is ?

Section B

Answer any five questions. Each carries two marks (5x2=10 marks)

16. What are PUSH() and POP() operations in stack?

17. What is infix notation? Give an example of an arithmetic expression in infix notation.

18. What is a Self-referential structure?

19. What is the significance of the NULL value in the link part of a node?

20. Differentiate between paths and trails.

21. Draw a full binary tree with at least 6 nodes.

22. What is the time complexity of an algorithm ?

23. What is the disadvantage of linear searching?

24. What do you mean by divide and conquer Algorithm

25. What is the use of Prim's algorithm

Section C

Answer any five questions. Each carries four marks (4x5=20 marks)

26. Differentiate static and dynamic memory allocation

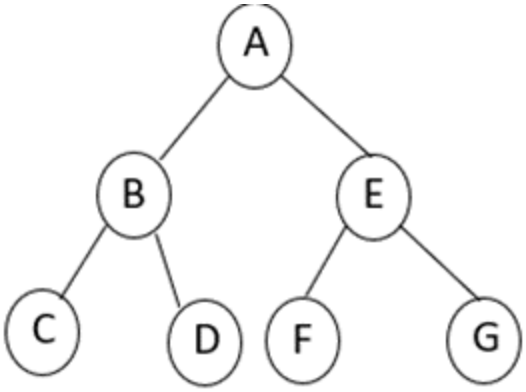
27. Explain enqueue and dequeue operations in detail using examples.

28. Write the algorithm for creating a linked list.

29. Explain the difference between a singly linked list and a doubly linked list.

30. A linked list of patients in the order of their token numbers is given.Write an algorithm that asks from the user to enter a token number of the patient which is desired to be deleted and then delete it from the list.

31. Consider the following binary tree:



Write the order of the nodes visited in:

A. In-order traversal

B. Pre-order traversal

C. Post-order traversal

32. What are the different ways of graph representation? Explain with an example.

33. Differentiate between directed and undirected graphs with examples.

34. Explain Insertion sorting with suitable example

35. Explain different asymptotic notations

Section D

Answer any two questions. Each carries fifteen marks (2x15=30 marks)

36. Describe how infix expressions are converted to postfix expressions? Convert the following infix expression into postfix expression

A - B - D * E / F + B * C

37. Describe the process of inserting nodes at
    a. The beginning
    b. At the end
    c. At a specific position in a singly linked list.

38. Simulate the result of inserting 50,55,60,15,10,40,20,45,30,70,80 into an empty AVL Tree.

39. Design and explain kruskal's algorithm for finding the minimum spanning tree of a graph.

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura
Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

Data Structures

COURSE CODE: B21CA04DC

ISBN 978-81-978764-7-9

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841